TYPED MEMORY MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

David Patrick Walker

January 2001

© David Patrick Walker 2001
ALL RIGHTS RESERVED

TYPED MEMORY MANAGEMENT

David Patrick Walker, Ph.D.
Cornell University 2001

Programming languages with sound static type systems have significant software engineering advantages over unsafe programming languages. Types can enforce a wide variety of program invariants at compile time, making it possible to catch programmer errors early in the software development cycle. Types can also improve programming language implementations by helping to guide program transformations and optimizations. In typed intermediate languages, such as the Java Virtual Machine Language, types certify crucial safety properties needed to implement secure and reliable software systems.

One area where traditional type systems provide little help to programmers is in the specification and enforcement of memory management invariants. In this thesis, I address this problem by developing new type systems with sophisticated mechanisms for checking programs that perform their own memory management. The central goal is to make it possible for programs to control memory resources, as in unsafe languages, and yet retain the software engineering benefits of strong type systems.

The central technical novelty of these type systems is the presence of a *static capability* that can control access to memory resources. Capabilities are instantiated in two ways in this thesis. First, they are instantiated with specifications of individual memory blocks. These capabilities make it possible to allocate, initialize, use and recycle data structures. They also provide a mechanism for controlling and reasoning about pointer aliasing. Second, capabilities are instantiated with vast memory regions (address spaces). This second alternative provides coarser-grained memory management as regions may contain many objects and all such objects are deallocated simultaneously when a region is freed.

For each of the resulting capability-based type systems I prove two theorems to quantify their memory management properties. First, I show that the type systems are sound: Well-typed programs never dereference dangling pointers or commit other memory faults at run time. Second, I prove a garbage collection theorem that specifies how effectively well-typed programs recycle memory.

Biographical Sketch

If occasionally somewhat overdressed, I make up for it by always being exceptionally overeducated.

– Oscar Wilde. The Importance of Being Earnest.

David Patrick Walker arrived on the scene on August 15th, 1972, just 43 days before Paul Henderson scored the goal. Paul Henderson's goal was ranked the eighth most significant event in Canadian history over the last 100 years; David's birth is surely only marginally less important. Please read any suitably well-respected book on Canadian history for a summary of David's accomplishments.

Acknowledgements

On reflection of my time at Cornell, I realize that my path to graduation has been remarkably trouble-free when compared with that of many other students. No doubt, my life has been made immeasurbly easier by the kindness and encouragement of the people that surround me. Whatever small amount of success I have had or might achieve is due mostly to them.

First of all, I would like to thank my parents for their love, support and teaching. My father started preparing me for grad school at an early age by explaining the role of "systems" in every-day life. Both Mum and Dad showed me the enjoyment of reading and exploring new ideas. When two people decide that the most important criterion for buying a house is the number of bookshelves it can contain, their son is destined for grad school. My brother is also very important to me. He does his best to make me a broader person and to school me in the fine arts of fun, relaxation, and, most importantly, sweet confabulation. Thanks Mark.

Greg Morrisett has been my mentor at Cornell and I simply cannot imagine where I would be without his encouragement, kindness and all-around smarts. He has a special kind of contagious enthusiasm that makes research exciting whenever he is in the room and his influence is evident in every line of this thesis. Outside the office, he is great friend who is always looking out for what is best for me.

I am also deeply endebted to many other faculty and staff here at Cornell. Dexter Kozen has always been able to give me wise advice whenever I ran into an academic problem. I am also grateful for all those evenings that he gave me a lift home after late-night hockey. Karla Consroe has made my days brighter with her smile and genuine good nature. Jan Baxter, Bob Constable, Andrew Meyers, Becky Personias, Fred Schneider, and Jim Wallis have all been a great help during my stay in graduate school.

I am sure I have learned more from my fellow students than from any classes or lectures I have ever attended. I have spent countless hours sipping coffee while Neal Glew patiently explained the technical intracacies of this or that. Dan Grossman picked up where Neal left off, both in terms of coffee supply and technical advice. Of equal importance, I can always count on Neal or Dan to share a beer at CTB or the Chapter House. Karl Crary has also been a wealth of ideas and has had a tremendous impact on my work. My officemates Stephanie Weirich and Nick Howe kindly kept me alive with continual input to, and (particularly in Nick's case) output from, the 5139 office cookie fund. They were also wonderful resources on everything from recursive kinds to how to turn off those damn Microsoft PowerPoint options to free food opportunities in the atrium. Steve Zdancewic has helped me with my writing on many occasions and he is always a dependendable bridge partner. It was always a pleasure to work and travel with Fred Smith. Riccardo Pucella read a draft of my thesis and made numerous helpful suggestions. I am remarkably lucky to have worked with so many talented, friendly people.

Finally, life in graduate school would have been unbearable without a core group of friends. I am happy that Gary Adams had the energy to drag me out to Cass Park hockey all those

frozen January nights and had the patience to suffer through so many Toronto Maple Leafs hockey games at other times. Mark Pearlman, a wizard in the kitchen, introduced me to the swiss-army bowl and the refridgerator's reversable door. Needless to say, my life will never be the same. Marcos Aguilera, Mark Hayden, Takako Hickey, Jason Hickey, Suzannah Hobbs, Amanda Holland-Minkley, Vera Kettnake, Tanya Morrisett, Ryan Budney's ultimate frisbee team and the Friday-night hockey group all helped make my time in grad school that much cheerier.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Type Systems . . . 2
  - 1.1.2 Types in Compilation . . . 3
  - 1.1.3 Tracing Garbage Collection . . . 4
- 1.2 Typed Memory Management . . . 5
- 1.3 Thesis Outline and Contributions . . . 6

2 Foundations: Linear Types **8**
- 2.1 A Linear Type System . . . 9
  - 2.1.1 Operational Semantics . . . 9
  - 2.1.2 Static Semantics . . . 12
  - 2.1.3 Store and Program Typing . . . 15
  - 2.1.4 Properties of the Type System . . . 15
  - 2.1.5 Intuitionistic Types . . . 17
- 2.2 Controlling Space Reuse . . . 19
  - 2.2.1 Explicit Allocation, Initialization and Reuse . . . 20
  - 2.2.2 Continuation-Passing Style . . . 22
- 2.3 Discussion . . . 28
  - 2.3.1 Related Work . . . 29
  - 2.3.2 Limitations . . . 30

3 Aliasing **31**
- 3.1 Informal Development . . . 31
  - 3.1.1 Abstraction Mechanisms . . . 34
- 3.2 Formal Syntax and Semantics . . . 37
  - 3.2.1 Type Constructors . . . 37
  - 3.2.2 Small Values and Expressions . . . 39
  - 3.2.3 Stores and Programs . . . 48
  - 3.2.4 Operational Semantics . . . 50
- 3.3 Properties of the Aliasing Language . . . 50
  - 3.3.1 Type Soundness . . . 50
  - 3.3.2 Complete Collection . . . 52
- 3.4 Applications . . . 52
  - 3.4.1 Destination-Passing Style . . . 52
  - 3.4.2 Stack Typing . . . 56
  - 3.4.3 Deutsch-Schorr-Waite Algorithms . . . 58

3.5 Discussion . . . 58
3.5.1 Arrays . . . 60
3.5.2 Related Work . . . 61
3.5.3 Limitations . . . 62

4 Regions **65**
4.1 Introduction to Region-based Memory Management . . . 66
4.2 The Capability Calculus . . . 68
4.2.1 Operational Semantics . . . 68
4.2.2 Types . . . 70
4.2.3 Store Types . . . 72
4.2.4 Expression Typing . . . 76
4.2.5 Non-linear Capabilities . . . 79
4.2.6 Run-time Values and Store Typing . . . 83
4.3 Properties of the Capability Calculus . . . 87
4.3.1 Type Soundness . . . 87
4.3.2 Complete Collection . . . 88
4.4 Examples . . . 88
4.5 Discussion . . . 90
4.5.1 Aggregate Data Structures . . . 92
4.5.2 Related Work . . . 93

5 Summary and Directions for Future Research **96**
5.1 Other Memory Management Strategies . . . 97
5.2 Towards More Expressive Safety Policies . . . 98
5.3 Conclusions . . . 99

A Alias Type Soundness & Garbage Collection Properties **100**

B Capability Calculus Type Soundness & Garbage Collection Properties **113**

Bibliography **129**

List of Figures

2.1 A Linear Language: Syntax 9
2.2 A Linear Language: Operational Semantics 11
2.3 A Linear Language: Static Semantics 13
2.4 A Linear Language: Copy & Free 18
2.5 A Linear CPS Language: Syntax 24
2.6 A Linear CPS Language: Operational Semantics 25
2.7 A Linear CPS Language: Static Semantics, Expressions 26
2.8 A Linear CPS Language: Static Semantics, expressions Cont. 27
2.9 A Linear CPS Language: Static Semantics, Values 27

3.1 Alias Types: Syntax, Kinds & Constructors 38
3.2 Alias Types: Well-formedness, Locations & Store Types 39
3.3 Alias Types: Well-formedness, Types 40
3.4 Alias Types: Syntax, Expressions 41
3.5 Alias Types: Static Semantics, Values 42
3.6 Alias Types: Static Semantics, Instructions 43
3.7 Alias Types: Static Semantics, Coercions 47
3.8 Alias Types: Syntax, Stores & Programs 48
3.9 Alias Types: Static Semantics, Storable Values 49
3.10 Alias Types: Operational Semantics, Programs 51
3.11 Alias Types: Operational Semantics, Coercions 52
3.12 Alias Types: Optimized Append 54
3.13 Deutsch-Schorr-Waite Tree Traversal 59
3.14 Deutsch-Schorr-Waite Tree Traversal, Cont. 60

4.1 Capabilities: Syntax 69
4.2 Capabilities: Operational Semantics 71
4.3 Capabilities: Static Semantics, Context Formation 72
4.4 Capabilities: Static Semantics, Type Formation 73
4.5 Capabilities: Static Semantics, Memory Types 74
4.6 Capabilities: Static Semantics, Linear Capability Equality 75
4.7 Capabilities: Static Semantics, Non-linear Capability Equality 79
4.8 Capabilities: Static Semantics, Subcapability Relation 82
4.9 Capabilities: Static Semantics, Heap and Word Values 84
4.10 Capabilities: Static Semantics, Declarations 85
4.11 Capabilities: Static Semantics, Expressions 86
4.12 Capabilities: Static Semantics, Memory 86
4.13 Capabilities: Static Semantics, Run-time Values 87

4.14 The Function `count` 89
4.15 Count with Efficient Memory Usage 91

Chapter 1

Introduction

Come, my friends,
'Tis not too late to seek a newer world.
Push off, and sitting well in order smite
The sounding furrows; for my purpose holds
To sail beyond the sunset, and the baths
Of all the western stars, until I die.
It may be that the gulfs will wash us down;
It may be we shall touch the Happy Isles,
And see the great Achilles, whom we knew.
Though much is taken, much abides; and though
We are not now that strength which in old days
Moved earth and heaven, that which we are, we are -
One equal temper of heroic hearts,
Made weak by time and fate, but strong in will
To strive, to seek, to find, and not to yield.

– *Alfred Lord Tennyson. Ulysses.*

This thesis sets out to discover new principles for constructing sound and flexible type systems for controlling computer resources. In particular, it explains how to manage memory resources safely. I will demonstrate that it is possible to grant programs explicit control over the allocation, use, reuse and deallocation of data structures, but automatically verify that these operations do not cause errors. This technology makes it possible for programmers to write space-efficient code while retaining the ability to reason and prove properties about the behaviour of their programs. In other words, I provide a foundation for the construction of secure and reliable software systems in an environment with limited resources.

1.1 Motivation

They were careless people [...] they smashed up things and creatures and then retreated back into their money or their vast carelessness, or whatever it was that kept them together, and let other people clean up the mess that they had made.

– *F. Scott Fitzgerald. The Great Gatsby.*

On September 23, 1999, NASA's Mars Climate Orbiter [NAS99a] fired its main engine before passing behind the red planet. NASA has received no further signals from the spacecraft; a three hundred and twenty-seven million dollar project was lost and has never been recovered. An independent board investigated this scientific disaster to determine the cause of the loss. One of the principal findings was that "the process to verify and validate certain engineering requirements and technical interfaces between some project groups, and between the project and its prime mission contractor, was inadequate." [NAS99b] When question about the incident, Dr. Edward Weiler, NASA's Associate Administrator for Space Science, explained.

> People sometimes make errors. The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft.

People cannot develop complex systems, such as the Mars Orbiter, without making mistakes. However, as Dr. Weiler suggests, errors can be detected, contained, and repaired if a sufficiently rich system of checks and balances are applied pervasively. Large-scale software systems are amongst the most complex human-conceived systems, and therefore, perhaps more than other sort of system, they can benefit from consistent use of redundant checking.

The root technical problem that befell the Mars Orbiter was a confusion between English units and metric units in ground-based navigation software. In essence, a failure to properly identify and process these two different *types* of data, the type of English unit calculations and the type of metric unit calculations, lead to a serious misunderstanding of the spacecraft's trajectory and the subsequent loss of communication and control.

Types arise naturally in every environment, not just multi-million dollar space programs. In the supermarket, food may be classified as fruit, bread, frozen, or canned. In mathematics, sets may be classified as sets of real numbers, sets of continuous functions, or sets of other sets. In computer programs, data may be classified as integers, characters, functions, or arrays. In many cases, failing to manipulate objects properly according to their type leads to irreparable system damage. For example, putting frozen items in the bread aisle causes food to thaw and spoil. Adding metric units to English units causes space ships to crash.

1.1.1 Type Systems

A *type system* is a collection of syntactic rules that specifies and constrains program behaviours. A *type checker* verifies that a program obeys the rules of the type system. In other words, it ensures that types have been correctly and consistently assigned to all parts of a program. This consistency checking makes type checkers extremely effective software engineering tools. Unlike full formal program verification, type checkers do not guarantee that programs compute correctly on all inputs. However, type checkers have the enormous practical advantage over verification tools because they can quickly and automatically check large programs for inconsistencies. In practice, by running a type checker over new software, a programmer can automatically detect a wide class of errors that commonly cause software systems to fail. For instance, a type checker can easily detect and reject a program that attempts to combine one object of type `EnglishUnits` with another object of type `MetricUnits` without first using a conversion function of type `EnglishUnits` $\to$ `MetricUnits` to mitigate the mismatch.

Type systems not only provide mechanical consistency checking, but also reasoning principles that programmers can use to understand their programs. For example, a type system can guarantee that program behaviour is independent of the representation of an abstract

type [Rey83]. This representation independence principle makes it possible to program modularly: A programmer can exchange one implementation of an abstraction for a more efficient one, assured that client modules cannot discern the difference and need not be changed. Types also provide structure that can simplify the process of proving properties of computations. In fact, simply by inspecting the type of a function, it is often possible to deduce many of its computational properties [Wad89]. Types also make it easy to construct logical relations that can be used to show deep properties of typed lambda calculi including termination [Sta85], information-flow [HR98] and countless others. Given the myriad of possibilities for programming with types, it is no wonder that the theory, design and implementation of type systems has been one of the most active fields of study in programming language research over the last forty years.

1.1.2 Types in Compilation

Relatively recently, a number of research groups recognized that it was both feasible and beneficial to propagate types present in source language programs into compiler intermediate languages [PJHH+93, TMC+96, MTC+96, LY96, SA95, DMTW97]. This idea gave rise to the notion of *type-directed* and *type-preserving* compilers. At each stage in the compilation process, a type-directed compiler has the option of using typing information to guide program transformations or optimizations. Of course, in order to make these transformations possible, a compiler must also be type-preserving; it must maintain typing information correctly for the stages that follow.

The same principles that allow programmers to reason soundly about well-typed source programs allow compiler-writers to reason about intermediate language programs: Types express invariants about program structure and a sound type system cuts down the number of contexts in which a program component can be executed. Compiler writers use these principles to guide program transformations and optimizations. For example, the Glasgow Haskell Compiler (GHC) uses typing information in its intermediate languages to direct optimizations including inlining and thunk-update avoidance among many others [TWM95, WP99]. The TIL compiler for Standard ML uses a typed intermediate language that makes it possible to construct and analyze typing information at run time as well as compile time [HM95, TMC+96]. This technology makes it possible to implement tag-free garbage collection, even in the presence of polymorphism, when accurate type information is not available at compile time. The FLINT compiler [SA95] has used types to direct sophisticated boxing and unboxing tranformations and the CHURCH project [DMTW97] has investigated how to use types to choose between closure representations.

Types also increase the reliability of compiled code. Many type-preserving compilers use an internal type checker to verify the consistency of intermediate language representations. Just as type checking source code can help detect programmer errors, type checking intermediate code can help detect bugs in a compiler. If a program type checks before but not after being run through a new optimization phase, then there must be an error in the compiler, provided the type checker is implemented correctly. In any event, this sort of redundant checking can certainly help to make compilers more robust.

An intermediate language type checker can also be used as a building block in a platform for secure mobile code—an idea that has been wildly successful for the Java virtual machine [LY96]. In this setting, Java programs are compiled into Java virtual machine language (JVML) applets in such a way that typing information is preserved. Before executing an untrusted JVML applet, a type checker ensures that the code is well-typed. Assuming the JVML type system is sound

and the JVML compiler and run-time system have been implemented correctly, any well-typed program will be safe to execute.

Typed assembly language (TAL) [MWCG98, MWCG99] and several other systems of proof-carrying code (PCC) [Nec97, NL98, Koz98] take this paradigm one step further by propagating typing information all the way through the compiler, beyond the level of traditional typed intermediate languages, and into executable binaries. More specifically, Cornell's typed assembly language project uses typing annotations to verify the safety of programs written for the Intel IA32 architecture. Necula and Lee have also built a compiler that targets the Intel IA32. They embed typing rules in a first-order logic and type-checking is implemented by checking a first-order logic proof. The principal advantage of this research is that it is now possible to exploit the benefits of types in all aspects of the implementation of modern programming languages.

1.1.3 Tracing Garbage Collection

Type-safe modern programming languages do not normally allow programmers to manage their own memory resources. Instead, they provide a run-time system that automatically reclaims memory using tracing garbage collection [Wil92]. In most circumstances, programming is difficult enough; there is no need to complicate the job by requiring the programmer to take care of the details of when, where and how to reuse memory resources. A correct tracing garbage collector can guarantee that all memory is eventually reclaimed (provided the program terminates) and it will never make the mistake of collecting memory too early.

Unfortunately, garbage collection incurs a significant computational cost. For example, in a study of Standard ML programs, Tarditi and Diwan found that storage management occupied anywhere from 19% to 46% of total execution time [TD96]. Some programs can afford to pay this cost, but for others, memory management can be a performance bottleneck. In the latter case, programmers need access to alternative memory management techniques. For example, the Ensemble distributed group communication system [Hay00] is written mostly in ML and therefore the primary memory management mechanism is a garbage collector. However, the implementers found that garbage collection was not the right technique for processing the large buffers used to store messages. After trying several custom memory management techniques, they finally found that a reference-counting scheme worked best. In the end, the cost of processing messages improved by a factor of 10 or more in some cases. Of course, in order to code this part of the system, they had to step outside the safety provided by ML and work in C.

Even when resources are not quite so scarce, the cost of garbage collection is significant enough to inspire many compiler writers to implement a vast array of optimizations to reduce allocation, reuse space, and consequently improve locality and speed up processing time. Unfortunately, the vast majority of these optimizations are studied in the context of an untyped compiler intermediate language. Consequently, in general, language implementations that produce fast code have not been able to take advantage of the reasoning principles, consistency checking, or security guarantees afforded by compiling with types.

Semantically, standard tracing garbage collection is also undesirable because it is a *meta-linguistic operation*. Garbage collection normally occurs "under the covers," outside standard semantic models of programming languages. Although some language definitions require certain space properties of their implementation, including efficient tail-call implementation [KCR98], in general, they make no requirements of the garbage collector [KCR98, MTHM97, LY96]. Consequently, it is often difficult to reason either formally or informally about the time and space behaviour of programs that rely on garbage collectors to recycle memory resources. A

particularly acute example of this problem occurred when Eva Tardos, teaching a group of Cornell undergraduates introductory complexity analysis, asked them to program quicksort in Java. Instead of seeing the predicted $n * log(n)$ curve for the algorithm, the students saw an n^2 curve due to excessive time spent in garbage collection.[1] Although, in this case, the difficulties could be attributed to a particularly poor garbage collection implementation, the problem remains that the cost of storage management is often ill-specified, and in practice, the space and time requirements of garbage-collected programs are difficult to estimate.

There are some theoretical and practical solutions to these problems. On the theoretical side, there has been recent research in space-profiling semantics [BG96, Min99], which can be used to prove memory consumption properties of ideal implementations. On the practical side, real-time garbage collectors [Bak78, BC99] (collectors with provable time and space properties) reduce pauses in execution time and can make space usage more predictable. However, both the theoretical models and the practical implementations normally assume that reachability defines garbage. In other words, an object is garbage if and only if there is no reference to it remaining in the program. Reachability is one *approximation* of garbage, but, as demonstrated by Morrisett, Felleisen and Harper [MH97], it is not the only one, nor is it necessarily always the best one. Morrisett *et al.* define garbage semantically as any heap-allocated object that influences the current computation. In the light of this definition, it is clear that many reachable objects may be garbage. Hence we should consider alternatives to standard trace-based collection.

Ideally, programmers should be able to select from a variety of memory management strategies. In the normal case, they will use standard garbage collection because it is hassle-free. However, when resources become scarce or it is necessary to reason carefully about program performance, programmers should have the option to use their choice of custom-tuned, application-specific memory management techniques. Moreover, in the best of all worlds, programmers (and compiler writers) should retain all the benefits of programming and compiling with types.

1.2 Typed Memory Management

Why don't type-safe languages allow explicit memory management?

Most type systems rely on the *Type Preservation* property to establish type safety [WF94]. In order to prove this property, we first define an abstract machine that runs programs in the language. We also define typing rules for the abstract machine states, derived from the typing rules for the programming language itself. Given this collection of typing rules, Type Preservation states that if a program or abstract machine state is well-typed then after one step in the computation the machine state will again be well-typed. A second property, *Progress*, guarantees that no well-typed program or abstract machine state will violate the safety policy on its next step. Together these two properties are sufficient to show that no well-typed program will violate the safety policy on *any* step of its computation. There are other techniques for establishing the type soundness of programs, but experience has shown that in many cases, this Preservation and Progress approach is the simplest.

Since Preservation insists that well-typed programs step to well-typed programs, the simplest way to achieve type safety is to insist that type structure change as little as possible from one step to the next. If we apply this idea to a language with heap allocation, we should

[1] The garbage collector was performing collections at fixed intervals. Therefore, an algorithm with expected $n * log(n)$ running time would perform on the order of $n * log(n)$ collections. Each collection involved scanning the live data, which included an array of size n. Therefore, the estimated running time was actually $n * log(n) * n$, and, in practice, it turned out to be n^2. Personal communication, Greg Morrisett. March 2000.

attempt to ensure that the type of the heap, and consequently the types of all objects within the heap, change as little as possible from one step to the next.

Principle 1 (Type Invariance) *To ensure type safety, every heap-allocated object should have one and only one type for the duration of program execution.*

Violating the type-invariance principle can easily give rise to an uns and afe language. For example, simple-minded attempts to mix polymorphism and references in ML-like languages quickly leads to unsoundness (*c.f.* Tofte [Tof90]).

The type-invariance principle provides the reason that most type-safe languages do not allow memory management operations such as explicit memory deallocation and recycling or user-level initialization of data structures. If a heap-allocated object has only one type during program execution then:

1. The object must have that type when it is allocated.

2. The object must continue to have that type after each evaluation step.

The first invariant implies that a user cannot allocate an object on the heap and later initialize the fields of the object. Allocation and initialization must occur together when the object is created. The second invariant implies that a user cannot explicitly recycle a heap-allocated object, using it to store values with a variety of types. It also implies that explicit deallocation is impossible, as deallocation implicitly allows memory to be later reused by a different part of the program and possibly at different types.

1.3 Thesis Outline and Contributions

In this thesis, I will challenge the type-invariance principle. By replacing this invariant with other memory managment invariants, I will show that it is possible to define a collection of sound type systems that allow explicit memory management operations including separate allocation and initialization, explicit memory reuse at different types, and memory deallocation. My goal is to allow programmers to express a variety of memory management invariants but to retain the software engineering, security, and reasoning principles provided by strongly typed programming languages.

In chapter two, I will lay the foundations for my work by introducing a simple linear type system. There is nothing particularly new in this system, but it introduces important concepts, particularly notions of store typing and the garbage collection properties of linear types. It serves as a point for comparison with more sophisticated type systems that I will develop later in this thesis.

Later in chapter two, I observe that while standard linear type systems specify the program points where data structures are deallocated, they actually give programmers little control over memory management. I devise a second language that uses similar type structure to standard linear languages, but gives programmers significantly more control over memory management. In particular, programmers have explicit control over allocation and deallocation routines and have the flexibility to reuse memory locations multiple times, so long as data structures remain unshared.

Chapter three demonstrates how to take a significant step beyond standard linear type systems. More precisely, I show how to allow a degree of aliasing within the context of a

sound type system and yet retain the ability to explicitly deallocate or recycle storage. The key technical contribution of this chapter is the introduction of *static capabilities* that control access to memory resources. In essence, these static capabilities add a level of indirection to the type system that models the level of indirection present in the run-time store. The initial definition of capabilities makes them relatively inflexible, but I resolve this problem by combining them with other type constructors including universal and existential polymorphic types, unions and recursive types. These type constructors interact elegantly together and provide a safe but flexible programming model. In order to demonstrate the flexibility of the new language, I show how to represent a number of data structures requiring aliasing in a type-safe way including cyclic and doubly-linked lists and trees. I also sketch how certain compiler data structures such as Pascal's displays can be encoded in the type system. Finally, I present a couple of space-efficient algorithms including *destination-passing style* algorithms [Wad85, Lar89, CO96] and Deutsch-Schorr-Waite or "link-reversal" traversal algorithms [SW67].

In chapter four, I introduce the notion of region-based memory management. A region is an area of memory capable of holding multiple objects. When a region is deallocated all objects in the region are also deallocated. Type theory for region-based memory management was originally developed by Tofte and Talpin [TT94, TT97]. On the surface, this memory management technique appears to bear no relation to the linear typing discipline developed in the previous chapters of this thesis. However, it turns out that it is possible to use capabilities to describe the regions that are accessible at any given program point. Moreover, the key to safe region deallocation is the possession of a linear capability. Still, linear capabilities are not sufficiently powerful to encode many region-based programs. Therefore, we introduce non-linear capabilities and a subtyping discipline that relates them to linear capabilities.

The observations made in this chapter not only illuminate an interesting theoretical connection between linear type systems and region-based memory management, but they also provide a provably sound principle for optimization. Unlike the original Tofte-Talpin system, region lifetimes in my language are not defined or restricted by the lexical structure of the program. Therefore, regions can be allocated in one lexical scope and then safely deallocated in another. This flexibility is very important in a practical region-based implementation.

One final contribution made in chapter four is a straightforward syntactic proof of soundness using the standard Type Preservation and Progress lemmas. Previous work on region-based type systems involved defining a monotonic operator on sets and using a greatest fixed-point construction [TT97].

In chapter five, I present directions for continuing research. The type systems that I have presented in this thesis are tuned to the problem of ensuring safe memory deallocation and reuse. However, similar techniques may be used to enforce other safety policies. I discuss some of these possibilities as well as further opportunities in the area of memory management.

Chapter 2

Foundations: Linear Types

> *I'M ALWAYS GOING TO BE PUBLISHED IN CAPITALS [...] BECAUSE IT WILL IMMEDIATELY GRAB THE READER'S ATTENTION.*
>
> – *John Irving. A Prayer for Owen Meany.*
>
> *DON'T PANIC!*
>
> – *Douglas Adams. A Hitch-Hiker's Guide to the Galaxy.*

In 1987, Girard developed linear logic [Gir87] in order to reason about computational resources. In this logic, every proposition is a valuable resource. Like an original Renoir or a carefully polished line of a Ph.D. thesis, a proposition cannot be duplicated nor can a proposition be thrown away — each one must be used exactly once. Girard's cleverness lies in the way he constructed the logical rules for this system. They act as a shrewd accountant, preventing both duplication and waste.

Computer scientists were quick to recognize that Girard's ideas had many applications in programming languages [Laf88, Abr93, Wad90, LM92, CGR96, TWM95, WP99, TW99]. Of particular relevance to this thesis is the fact that linear logic provides an elegant foundation for controlling the memory resources used by programs. Object construction incurs a cost in the form of memory resources and if these resources are not managed carefully, programs will make inefficient use of memory. *Linear type systems* attempt to curb these inefficiencies by ensuring each object is used exactly once and making it possible to recover an object's memory after this single use.

This chapter explores the capacity of linear types to encode memory management invariants. The bulk of the chapter is review so those familiar with linear and intuitionistic type systems, operational and particularly allocation-style semantics, and continuation-passing style could certainly afford to skip this chapter and move on to the next. On the other hand, the development here may be helpful even to experts as it introduces some of the notation that is used later. It should also help to clarify the relationship between standard linear type systems and the more advanced type systems of future chapters.

In section 2.1, I briefly review the semantics of a standard linear type system, drawing on the exposition by Turner and Wadler [TW99] for the main technical ideas. The main goal of the review is to introduce the reader to the notion of linear typing, which is fundamental for understanding later results, and to provide a concrete starting point to begin our exploration of typed memory management.

types	τ	$::=$	$bool \mid \tau_1 \otimes \tau_2 \mid \tau_1 \multimap \tau_2$
expressions	e	$::=$	$x \mid b \mid \texttt{if}\ e_1\ (e_2 \mid e_3) \mid \langle e_1, e_2 \rangle \mid \texttt{let}\, x, y = e_1\, \texttt{in}\, e_2 \mid$ $\lambda x{:}\tau.e \mid e_1 e_2$

Figure 2.1: A Linear Language: Syntax

In section 2.2, I analyze the linear type system I have just explained and observe that programmers still have little control over how or when memory is reused. These observations lead me to develop a modified calculus that gives programmers significantly greater control over memory management decisions. The modifications are not particularly deep and I do not consider this language one of my main contributions, but it helps to bridge the gap between linear type systems and the more advanced results in later chapters. To help facilitate the transition to the new work, the modified language is presented in continuation-passing style. This decision is not essential here or in the later chapters but it simplifies certain technical difficulties.

The last section of this chapter provides further information on related work on linear type systems.

2.1 A Linear Type System

Figure 2.1 defines the syntax of a call-by-value, linear lambda calculus with booleans (b is either `true` or `false`), linear pairs ($\langle e_1, e_2 \rangle$) with type $\tau_1 \otimes \tau_2$ and functions ($\lambda x{:}\tau.e$) with type $\tau \multimap \tau'$. Throughout this thesis, I will use letters at the upper end of the alphabet (x, y, z, w, ...) to denote term-level variables, which are drawn from a countably infinite set `Var`. Expressions (and later types) that differ only in the names of bound variables are considered equivalent. The free variables of an expression ($\mathrm{FV}(e)$) are defined in the ordinary way.

The elimination forms include the if statement ($\texttt{if}\ e_1\ (e_2 \mid e_3)$), an operation for extracting both components of a pair ($\texttt{let}\, x, y = e_1\, \texttt{in}\, e_2$) and standard function application ($e_1 e_2$).

2.1.1 Operational Semantics

Following Turner and Wadler's development [TW99], I will use an allocation semantics [Lau93, MFH95, MH97] to make the meaning of the linear lambda calculus terms precise. The defining characteristic of an allocation semantics is the presence of a *store* that maps locations (memory addresses) to values. Programs refer to stored values indirectly through locations rather than referring to stored values directly in the program text, as in a standard substitution-based semantics. When the same address appears twice in a program (or in a data structure) the two occurrences are called *aliases* of one another and the associated stored value is said to be *shared* between the two program parts.

The primary advantage of an allocation semantics is that it makes sharing explicit. The standard operational models of the lambda calculus are too high-level to capture sharing, and therefore it is extremely difficult to use these models to reason about memory management properties. For example, consider a pair of pairs. In the standard, substitution-based model of the lambda calculus, this data structure might have the following form:

$$\langle\langle 2,3\rangle,\langle 2,3\rangle\rangle$$

In this semantics, it is impossible to distinguish between the data structure in which the underlying pairs are shared and the data structure in which the underlying pair are unshared. In other words, does the first component of the pair occupy the same memory location as the second component? Both pairs have the shape $\langle 2,3\rangle$, so they might share, but we do not know for sure.

In contrast, in an allocation semantics, we can easily distinguish between shared and unshared data structures. As stated above, the store represents the location at which each object is allocated, and objects are referred to indirectly via these locations. Therefore to distinguish between shared and unshared data structures, we need to determine whether locations are equal or not. For instance, assume the pair is allocated at address x. In this case, a store that shares the two components of x has the form:

$$\{x \mapsto \langle y,y\rangle, y \mapsto \langle 2,3\rangle\}$$

whereas a store that does not share the two components of x has the form:

$$\{x \mapsto \langle y,z\rangle, y \mapsto \langle 2,3\rangle, z \mapsto \langle 2,3\rangle\}$$

The first operational models of linear type systems (work by Lafont [Laf88] and some of Abramsky's early research) did not use an allocation semantics and, naturally, there was considerable confusion as to the memory management properties that they possessed.

Formal Semantics In this chapter, locations are modelled as variables and the *storable values* (sometimes called *heap values*) are a subset of the expressions:

$$\textit{storable values} \quad h \quad ::= \quad b \mid \langle x_1, x_2\rangle \mid \lambda x{:}\tau.e$$

For the sake of uniformity, all objects in the language are allocated in the store, even "small" values, like booleans, that would not be heap-allocated in a practical space-conscious implementation. Since all values are allocated in the store, the components of a pair, x_1 and x_2, are always addresses.

A *store* (S) is a finite partial map from variables to storable values. I use the notation $\{x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\}$ to denote finite partial maps. $Dom(S)$ denotes the domain of a map; $Rng(S)$ denotes the codomain. The notation $S\backslash x$ denotes a store S' that is undefined at x, but is otherwise identical to S. Finally, when $Dom(S) \cap Dom(S') = \emptyset$, the notation SS' denotes a new store S'' such that

$$S''(x) = \begin{cases} h & \text{if } S(x) = h \\ h & \text{if } S'(x) = h \end{cases}$$

Figure 2.2 presents the formal operational semantics, which maps programs to programs, where a program (P) is a store paired with an expression to be evaluated. I use the notation $\mathsf{P} \longmapsto_L \mathsf{P}'$ to denote a single operational step and $\mathsf{P} \longmapsto_L^* \mathsf{P}'$ for the reflexive, transitive closure of the single-step semantics. The presentation from of the semantics is simplified in the standard way through the use of evaluation contexts E, which are expressions with a single hole ($[\,]$) that can be filled by any beta-redex. These contexts fix the order of evaluation to be left-to-right and

$$\begin{array}{llll} \textit{evaluation ctxts} & E & ::= & [\,] \mid \texttt{if}\ E\ (e_1 \mid e_2) \mid \langle E, e\rangle \mid \langle x, E\rangle \mid \texttt{let}\ x, y = E\ \texttt{in}\ e \mid \\ & & & E\ e \mid x\ E \end{array}$$

$$\frac{(S, e) \longmapsto_L (S', e')}{(S, E[e]) \longmapsto_L (S', E[e'])}\ (\texttt{LO-context})$$

$$(S, b) \longmapsto_L (S\{x \mapsto b\}, x) \quad \text{where } x \notin \mathit{Dom}(S) \qquad (\texttt{LO-bool})$$

$$(S\{x \mapsto \texttt{true}\}, \texttt{if}\ x\ (e_1 \mid e_2)) \longmapsto_L (S, e_1) \qquad (\texttt{LO-ift})$$

$$(S\{x \mapsto \texttt{false}\}, \texttt{if}\ x\ (e_1 \mid e_2)) \longmapsto_L (S, e_2) \qquad (\texttt{LO-iff})$$

$$(S, \langle x_1, x_2\rangle) \longmapsto_L (S\{x \mapsto \langle x_1, x_2\rangle\}, x) \quad \text{where } x \notin \mathit{Dom}(S) \qquad (\texttt{LO-pair})$$

$$(S\{x \mapsto \langle x_1, x_2\rangle\}, \texttt{let}\ y_1, y_2 = x\ \texttt{in}\ e_2) \longmapsto_L (S, e_2[x_1, x_2/y_1, y_2]) \qquad (\texttt{LO-proj})$$

$$(S, \lambda y{:}\tau.e) \longmapsto_L (S\{x \mapsto \lambda y{:}\tau.e\}, x) \quad \text{where } x \notin \mathit{Dom}(S) \qquad (\texttt{LO-fun})$$

$$(S\{x_1 \mapsto \lambda y{:}\tau.e\}, x_1\, x_2) \longmapsto_L (S, e[x_2/y]) \qquad (\texttt{LO-app})$$

Figure 2.2: A Linear Language: Operational Semantics

call-by-value. The notation $X[Y_1, \ldots, Y_n/x_1, \ldots, x_n]$ denotes the simultaneous capture-avoiding substitution of $Y_1, \ldots, Y_n$ for $x_1, \ldots, x_n$ in X.

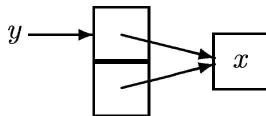
Whenever a program introduces a value, the run-time system allocates it in the store and after any use, the value is immediately deallocated. For example, the pair introduction rule, `LO-pair`, allocates the value $\langle x_1, x_2 \rangle$ at a fresh location x in the store. The pair elimination rule, `LO-proj`, projects the two components of the pair x_1 and x_2, substitutes the components for the bound variables y_1 and y_2 in the body of the expression and removes the binding $\{x \mapsto \langle x_1, x_2 \rangle\}$ from the store. The rules for booleans and functions are similar in their use of the store.

2.1.2 Static Semantics

Since the operational semantics deallocates an object immediately after using it, the static semantics must ensure that each object is used just once. Any attempt to use an object multiple times will cause a program to crash due to the fact that an object is automatically deallocated after its first use. In order to enforce the use-once condition, the linear type system I will develop will impose the additional constraint that there is only one pointer to any object.[1] In other words, it disallows aliasing. In order to see some of the potential dangers that can arise when pointers are allowed to alias one another, consider the function `pair`:

$$\texttt{pair} = \lambda x{:}\tau.\langle x, x \rangle$$

The function makes a shallow copy of its argument and returns a pair y with the following shape:



Both elements of the pair are aliases of one another and if we are not careful, they could lead to trouble. Once one pointer is used, the other will *dangle*. In other words, it will point to unallocated storage and using it will cause the program to crash:

```
let y, z = pair(x) in
⟨use(y),              % use the first component
 use(z)⟩              % use the dangling pointer & crash
```

One of the goals of linear type systems is to rule out these sources of failure. A second goal of the linear type system is to prevent the creation of garbage. Under certain circumstances, the `discard` function is guilty of the second sin:

$$\texttt{discard} = \lambda x{:}\tau.\texttt{true}$$

If the only pointer to x is passed to the `discard` function then x will be unreachable after the function call. In order to recycle x, we would have to rely on meta-linguistic support mechanisms such as a tracing garbage collector to do it for us. Linear type systems make it unnecessary to rely on this external support by rejecting such programs at compile time.

[1] Notice the distinction between the number of uses of an object and the number of pointers to an object. Even if an object can only be used once, it may still be safe to allow multiple pointers to the object. However, this is not the case in the linear type system we are studying here.

$\boxed{\Gamma \vdash_L e : \tau}$

$$\frac{}{x{:}\tau \vdash_L x : \tau} \; (\texttt{L-var})$$

$$\frac{}{\cdot \vdash_L b : \mathit{bool}} \; (\texttt{L-bool})$$

$$\frac{\Gamma_1 \vdash_L e_1 : \mathit{bool} \quad \Gamma_2 \vdash_L e_2 : \tau \quad \Gamma_2 \vdash_L e_3 : \tau}{\Gamma_1, \Gamma_2 \vdash_L \texttt{if}\; e_1\; (e_2 \mid e_3) : \tau} \; (\texttt{L-if})$$

$$\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \quad \Gamma_2 \vdash_L e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_L \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \; (\texttt{L-pair})$$

$$\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \otimes \tau_2 \quad \Gamma_2, x{:}\tau_1, y{:}\tau_2 \vdash_L e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash_L \texttt{let}\, x, y = e_1 \,\texttt{in}\, e_2 : \tau_3} \; (x, y \notin \mathit{Dom}(\Gamma_2)) \; (\texttt{L-proj})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash_L e : \tau_2}{\Gamma \vdash_L \lambda x{:}\tau_1.e : \tau_1 \multimap \tau_2} \; (x \notin \mathit{Dom}(\Gamma)) \; (\texttt{L-fun})$$

$$\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash_L e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash_L e_1 e_2 : \tau_2} \; (\texttt{L-app})$$

$$\frac{\Gamma_1, x_2{:}\tau_2, x_1{:}\tau_1, \Gamma_2 \vdash_L e : \tau}{\Gamma_1, x_1{:}\tau_1, x_2{:}\tau_2, \Gamma_2 \vdash_L e : \tau} \; (\texttt{L-exchange})$$

Figure 2.3: A Linear Language: Static Semantics

The standard linear type system is remarkably simple and yet it is able to prevent both the dangerous `pair` function and the wasteful `discard`. The main judgement in the system has the form $\Gamma \vdash_L e : \tau$. This judgement states that in the context Γ, expression e is well-formed and has type τ. The context Γ is either empty $(\cdot)$ or it is a list of variable-type bindings: $(x_1{:}\tau_1, \ldots, x_n{:}\tau_n)$. The notation $Dom(\Gamma)$ denotes the set of first-components of the bindings in Γ.[2]

The typing rules for the language appear in Figure 2.3. These rules are designed to maintain strict control over the context Γ and through this mechanism ensure that every object is used exactly once. For example, in the typing rule for variables, `L-var`, the type system requires that the context Γ contain exactly one binding, the binding for the variable expression itself. The rule for booleans, `L-bool`, requires there be no bindings in the context at all. If either of these rules allowed the context to contain multiple variable bindings where one (in the case of `L-var`) or zero (in the case of `L-bool`) bindings are actually used then some bindings would go unused and the program would create garbage.

For more complex expressions that contain several subexpressions, the context is split into a number of disjoint parts that are used to check each subexpression separately. The notation Γ_1, Γ_2 on the right-hand side of a turnstyle indicates a nondeterministic split of the context. For example, examine the rule `L-pair` for linear pairs. It splits the context into two parts, Γ_1 and Γ_2, and each part is used separately to check the two subexpressions e_1 and e_2. Here is a sample derivation:

$$\dfrac{\dfrac{\dfrac{}{y{:}\tau_2 \vdash_L y : \tau_2}\;(\texttt{L-var}) \qquad \dfrac{}{x{:}\tau_1 \vdash_L x : \tau_1}\;(\texttt{L-var})}{y{:}\tau_2, x{:}\tau_1 \vdash_L \langle y, x\rangle : \tau_2 \otimes \tau_1}\;(\texttt{L-pair})}{x{:}\tau_1, y{:}\tau_2 \vdash_L \langle y, x\rangle : \tau_2 \otimes \tau_1}\;(\texttt{L-exchange})$$

In the first line of this derivation, I use `L-exchange` to reorder the elements in the context. In many derivations, `L-exchange` will have to be used many times to sort the context. Fortunately, however, there is no cost to reordering typing assumptions. For this reason, in future type systems, I will omit the exchange rule and implicitly treat contexts that differ only in the ordering of their elements as equivalent.

In terms of the role of the context, the rules `L-proj` and `L-app` are quite similar to `L-pair`. The rule `L-if` splits the context into two parts, Γ_1 and Γ_2. The second part is used to check both e_2 and e_3. Since only one of the two branches of an if statement are ever executed, it is safe to use Γ_2 to check both. As usual, the rule for functions, `L-fun`, appends a single copy of the binding $x{:}\tau$ to the context.

Examples of programs that *do not* type check are often more instructive than those that do. In this case, we can try to use the linear typing rules to verify the `pair` function:

$$\dfrac{\dfrac{\dfrac{}{x{:}\tau \vdash_L x : \tau}\;(\texttt{L-var}) \qquad \dfrac{}{\cdot \vdash_L x : \tau}\;(\texttt{Wrong})}{x{:}\tau \vdash_L \langle x, x\rangle : \tau \otimes \tau}\;(\texttt{L-pair})}{\cdot \vdash_L \lambda x{:}\tau.\langle x, x\rangle : \tau \multimap \tau \otimes \tau}\;(\texttt{L-fun})$$

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash_L x : \tau}\;(\texttt{Wrong}) \qquad \dfrac{}{x{:}\tau \vdash_L x : \tau}\;(\texttt{L-var})}{x{:}\tau \vdash_L \langle x, x\rangle : \tau \otimes \tau}\;(\texttt{L-pair})}{\cdot \vdash_L \lambda x{:}\tau.\langle x, x\rangle : \tau \multimap \tau \otimes \tau}\;(\texttt{L-fun})$$

[2]Later, I will treat Γ as a finite partial map and implicitly assume these maps are equivalent up to reordering of their elements. However, for development of the semantics of the linear type system, I prefer to introduce the structural rules for the context explicitly.

We quickly discover that there is no way to divide up the context in the (`L-pair`) rule to check both occurrences of the variable x. Difficulties also arise when attempting to check the `discard` function. This time, however, rather than having too few bindings in the context, we have too many bindings and we are unable to apply the (`L-bool`) rule:

$$\dfrac{\dfrac{}{x{:}\tau \vdash_L \texttt{true} : \mathit{bool}}\ (\texttt{Wrong})}{\cdot \vdash_L \lambda x{:}\tau.\texttt{true} : \tau \multimap \mathit{bool}}\ (\texttt{L-fun})$$

2.1.3 Store and Program Typing

The store typing rules capture memory management invariants that must be preserved by each step of a computation. In the case of a linear type system, the central invariant is that there must be no aliasing. This invariant is expressed in a judgement with the form $\vdash_L S : \Gamma$. The two rules for store typing follow.

$$\dfrac{}{\vdash_L \{\} : \cdot}\ (\texttt{L-store-empty})$$

$$\dfrac{\vdash_L S : \Gamma_1, \Gamma_2 \quad \Gamma_1 \vdash_L h : \tau}{\vdash_L S\{x \mapsto h\} : \Gamma_2, x{:}\tau}\ (x \notin \mathit{Dom}(S))\ (\texttt{L-store})$$

The first rule states that the empty store is well formed and can be described by the empty context. The second rule is reminiscent of expression typing rules, such as `L-pair`, that split the context into disjoint parts to verify separate subexpressions. Here, if a store S can be described by the context Γ_1, Γ_2 then the first subcontext (Γ_1) is to be used in verifying the next value in the store (h) and the result is described by ($\Gamma_2, x{:}\tau$). Unlike the store typing rules we will see later, these rules build up a store typing inductively. These rules, together with the expression typing rules (used to check $\Gamma_1 \vdash_L h : \tau$) guarantee that it is impossible to create sharing or cycles in the store.

In order for a whole program $P = (S, e)$ to be well-formed, all of the store locations must be used during the execution of e. In order for the environment (the operating system that runs the program, for instance) to interpret the final result of the computation properly, I also require that the result inhabit a distinguished answer type and for simplicity, the answer type will be *bool*.[3] Hence, to type closed programs, we use the rule

$$\dfrac{\vdash_L S : \Gamma \quad \Gamma \vdash_L e : \mathit{bool}}{\vdash_L (S, e)}\ (\texttt{L-prog})$$

2.1.4 Properties of the Type System

As described in the introduction, the simplest way to prove several significant memory management properties of the linear type system is to adapt the syntactic proof techniques popularized by Wright and Felleisen [WF94]. I will not plod through all the details such proofs here as similar results have been proven elsewhere in the literature already (see Turner and Wadler [TW99]). I will give the central definitions and lemmas as the same concepts reappear in the more challenging work later in this thesis.

Type Soundness, which informally states that well-formed programs "won't go wrong," is the first important property to prove. To "go wrong" means to step to one of the bad or *stuck* programs that is not a terminal state, but for which no operational rule applies:

[3] This restriction is inessential, but it makes some of the definitions below and analogous ones in the next chapter slightly more elegant.

Definition 2 (Linear Stuck Program) *A program* $\mathtt{P}$ *is* stuck *if it does not satisfy either of the two following constraints:*

1. *There exists a* $\mathtt{P}'$ *such that* $\mathtt{P} \longmapsto_L \mathtt{P}'$.
2. $\mathtt{P}$ *is a well-formed terminal program:* $\mathtt{P} = (S, x)$ *and* $S(x) = b$.

For example, $(\{x \mapsto \mathtt{true}, x' \mapsto \mathtt{false}\}, x\,x')$ is stuck because it is not a terminal program and the only operational rule that might apply is the application rule `L0-app`, but `L0-app` requires that location x hold a closure, which it does not.

Having defined the stuck programs, type soundness can be summarized by saying that well-formed programs do not evaluate to stuck programs:

Proposition 3 (Linear Type Soundness) *If* $\vdash_L \mathtt{P}$ *and* $\mathtt{P} \longmapsto_L^* \mathtt{P}'$ *then* $\mathtt{P}'$ *is not stuck.*

Type Soundness may be proven by induction on the length of the reduction sequence $\mathtt{P} \longmapsto_L^* \mathtt{P}'$. The proof relies directly on two lemmas, Progress, which states that well-formed programs are not stuck, and Subject Reduction, which states that well-formed programs only reduce to well-formed programs.

Lemma 4 (Linear Progress) *If* $\vdash_L \mathtt{P}$ *then* $\mathtt{P}$ *is not stuck.*

Lemma 5 (Linear Subject Reduction) *If* $\vdash_L \mathtt{P}$ *and* $\mathtt{P} \longmapsto_L \mathtt{P}'$ *then* $\vdash_L \mathtt{P}'$.

Type Soundness, as I have defined it, captures half the memory management properties of the linear type system. It implies that the run-time system does not deallocate objects too early since for any evaluation context E,

- $(S, E[z\,y])$
- $(S, E[\mathtt{let}\,x, y = z\,\mathtt{in}\,e])$
- $(S, E[\mathtt{if}\ z\ (e_1 \mid e_2)])$

are all stuck when $z \notin \mathit{Dom}(S)$. However, since the definition of stuck programs is rather weak, the proposition says nothing about the linear type system's capacity to rule out the creation of garbage.

A common definition of garbage is based on the idea of *reachability*. As illustrated by Morrisett, Felleisen and Harper [MFH95], reachability is not the only definition of garbage. In general, it is only one approximation of a more semantic notion of garbage. I use it here as it characterizes the garbage collection capabilities of the linear type system effectively yet nicely contrasts those of the type systems to come.

Definition 6 (Reachability)

$$\begin{array}{rcl} \mathit{Reachable}(S, e) & = & \mathit{Reachable}(S, \mathrm{FV}(e)) \\ \mathit{Reachable}(S, X) & = & X \cup \bigcup_{x \in X} \mathit{Reachable}(S, S(x)) \end{array}$$

Definition 7 (Reachability-Based Garbage) *A binding* $\{x \mapsto h\}$ *is garbage with respect to a program* $(S\{x \mapsto h\}, e)$, *if* $x \notin \mathit{Reachable}(S\{x \mapsto h\}, e)$

We can easily prove that well-formed linear programs create no reachability-based garbage, a property I call *complete collection*:

Proposition 8 (Linear Complete Collection) *If* $\vdash_L \mathtt{P}$ *and* $\mathtt{P} \longmapsto_L^* \mathtt{P}'$ *then* $\mathtt{P}'$ *contains no reachability-based garbage.*

The proof follows from Subject Reduction and the fact that well-formed programs contain no reachability-based garbage:

Lemma 9 *If* $\vdash_L \mathtt{P}$ *then* $\mathtt{P}$ *contains no reachability-based garbage.*

2.1.5 Intuitionistic Types

As always, the rules that are left out of a type system play at least as large a role as the rules that have been put in. In this case, two rules, in particular, have been intentionally omitted:

$$\frac{\Gamma, x{:}\tau, x{:}\tau \vdash_I e : \tau'}{\Gamma, x{:}\tau \vdash_I e : \tau'}\ (\texttt{I-contraction}) \qquad \frac{\Gamma \vdash_I e : \tau'}{\Gamma, x{:}\tau \vdash_I e : \tau'}\ (\texttt{I-weakening})$$

An intuitionistic type system for the simply-typed lambda calculus can be formulated with `I-contraction` and `I-weakening` and the same rules for variables, booleans, pairs and functions as in the linear type system. In order to avoid confusion between intuitionistic and linear systems, I will prefix rule names with `I` rather than `L`, as in `I-var`, `I-bool`, etc. when working in the intuitionistic setting. I use the standard arrow ($\rightarrow$) and product ($\times$) symbols for intuitionistic function and pair types. In this new type system, it possible to type the `pair` function. The `I-contraction` rule makes all the difference:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x{:}\tau \vdash_I x : \tau}\ (\texttt{I-var}) \quad \cfrac{}{x{:}\tau \vdash_I x : \tau}\ (\texttt{I-var})}{x{:}\tau, x{:}\tau \vdash_I \langle x, x\rangle : \tau \times \tau}\ (\texttt{I-pair})}{x{:}\tau \vdash_I \langle x, x\rangle : \tau \times \tau}\ (\texttt{I-contraction})}{\cdot \vdash_I \lambda x{:}\tau.\langle x, x\rangle : \tau \rightarrow \tau \times \tau}\ (\texttt{I-fun})$$

Similarly, `I-weakening` makes it possible to type `discard`:

$$\cfrac{\cfrac{\cfrac{}{\cdot \vdash_I \texttt{true} : \mathit{bool}}\ (\texttt{I-bool})}{x{:}\tau \vdash_I \texttt{true} : \mathit{bool}}\ (\texttt{I-weakening})}{\cdot \vdash_I \lambda x{:}\tau.\texttt{true} : \tau \rightarrow \mathit{bool}}\ (\texttt{I-fun})$$

The intuitionistic type system, being more lenient than the linear type system, does not provide the same memory management properties. Data structures can clearly be used multiple times, leading to aliasing that is not tracked in the type system. Hence, it is unsafe to deallocate intuitionistic data structures and, in general, a garbage collector must run in the background to recycle memory.

On the other hand, the fact that an intuitionistic type system accepts programs such as `pair` and `discard` suggests that type-safe languages with implicit garbage collection may be more expressive than type-safe languages that explicitly manage memory. Indeed, the linear language described so far is unable to encode these programs. However, the linear language is easily extended with primitives that express the computational content of the contraction and weakening rules:

$$\textit{reduction ctxts} \quad E \quad ::= \quad \cdots \mid \texttt{let}\, x, y = \texttt{deepcopy}(E)\, \texttt{in}\, e_2 \mid \texttt{deepfree}(E); e_2$$

$$\begin{array}{ll} (S_1\{x_1 \mapsto h_1\}, \texttt{let}\, y_1, y_2 = \texttt{deepcopy}(x_1)\, \texttt{in}\, e) \longmapsto_L & (\texttt{LO-dcopy}) \\ \quad (S_1\{x_1 \mapsto h_1\} S_2\{x_2 \mapsto h_2\}, e[x_1, x_2/y_1, y_2]) & \\ \quad \text{where } (S_2, x_2, h_2) = \mathit{copy}(S_1, h_1, \texttt{Var} - (\mathit{Dom}(S_1) \cup \{x_1\})) & \end{array}$$

$$\begin{array}{ll} (S\{x \mapsto h\}, \texttt{deepfree}(x); e_2) \longmapsto_L (S', e_2) & (\texttt{LO-dfree}) \\ \quad \text{where } S' = \mathit{free}(S\{x \mapsto h\}, x, h) & \end{array}$$

$$\begin{array}{llcl} \multicolumn{4}{l}{\mathit{copy}(S, h, L) = (S'_1 \cdots S'_n\{x'_1 \mapsto h'_1, \ldots, x'_n \mapsto h'_n\}, x', h')} \\ \text{where} & x_1, \ldots, x_n & = & \mathrm{FV}(h) \\ & L_1, \ldots, L_n & = & \text{a partition of } L \text{, each } L_i \text{ countably infinite} \\ & (S'_i, x'_i, h'_i) & = & \mathit{copy}(S, S(x_i), L_i) \text{ for } 1 \le i \le n \\ & x' & \in & L - \bigcup_{i \in 1,\ldots,n} \mathit{Dom}(S_i) \cup \{x'_i\} \\ & h' & = & h[x'_1, \ldots, x'_n / x_1, \ldots, x_n] \end{array}$$

$$\begin{array}{llcl} \multicolumn{4}{l}{\mathit{free}(S_0, x, h) = S_{n+1}} \\ \text{where} & x_1, \ldots, x_n & = & \mathrm{FV}(h) \\ & S_i & = & \mathit{free}(S_{i-1}, x_i, S_0(x_i)) \text{ for } 1 \le i \le n \\ & S_{n+1} & = & S_n \backslash x \end{array}$$

$$\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \quad \Gamma_2, x{:}\tau_1, y{:}\tau_1 \vdash_L e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_L \texttt{let}\, x, y = \texttt{deepcopy}(e_1)\, \texttt{in}\, e_2 : \tau_2}\ (x, y \notin \mathit{Dom}(\Gamma_2))\ (\texttt{L-dcopy})$$

$$\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \quad \Gamma_2 \vdash_L e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_L \texttt{deepfree}(e_1); e_2 : \tau_2}\ (\texttt{L-dfree})$$

Figure 2.4: A Linear Language: Copy & Free

$$\textit{expressions} \quad e \quad ::= \quad \cdots \mid \texttt{let}\, x, y = \texttt{deepcopy}(e_1)\, \texttt{in}\, e_2 \mid \texttt{deepfree}(e_1); e_2$$

The `deepcopy` primitive implements the contraction rule by making a complete copy of its argument. No parts of the copy are shared with the original and both are used in the rest of the program. The `deepfree` primitive implements the weakening rule. It deallocates its argument and all of its substructures.

Figure 2.4 presents the extended operational and static semantics for the linear language with `deepcopy` and `deepfree`. The operational rule for deep copies, `LO-dcopy`, duplicates the subtree rooted at its argument location x. It depends on the auxiliary function *copy*, which is careful to select fresh addresses for the new tree that do not appear elsewhere in the store. The deep free rule, `LO-dfree`, deletes the subtree rooted at the location x; it relies on the auxiliary function *free* to do most of the work. In the static sematics, the copy rule consumes the context Γ_1 when checking e_1 and then adds bindings for x and y to Γ_2 when verifying e_2. The rule

`L-dfree` uses the entire context Γ to check the single subexpression e.

Given a type-safe intuitionistic program, it is always possible to construct an equivalent type-safe linear program. Whenever the intuitionistic type system uses `I-contraction`, it can be replaced with the `L-dcopy` rule. Likewise, `I-weakening` can be replaced by `L-dfree`. For example it is possible to define and verify a `linear-pair` function:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y{:}\tau \vdash_L y : \tau}\ (\texttt{L-var}) \quad \dfrac{}{z{:}\tau \vdash_L z : \tau}\ (\texttt{L-var})}{y{:}\tau, z{:}\tau \vdash_L \langle y, z\rangle : \tau \otimes \tau}\ (\texttt{L-pair})}{x{:}\tau \vdash_L \texttt{let}\, y, z = \texttt{deepcopy}(x)\, \texttt{in}\, \langle y, z\rangle : \tau \otimes \tau}\ (\texttt{L-dcopy})}{\cdot \vdash_L \lambda x{:}\tau.\texttt{let}\, y, z = \texttt{deepcopy}(x)\, \texttt{in}\, \langle y, z\rangle : \tau \multimap \tau \otimes \tau}\ (\texttt{L-fun})$$

as well as a `linear-discard` function:

$$\dfrac{\dfrac{\dfrac{}{x{:}\tau \vdash_L x : \tau}\ (\texttt{L-var}) \quad \dfrac{}{\cdot \vdash_L \texttt{true} : \mathit{bool}}\ (\texttt{L-bool})}{x{:}\tau \vdash_L \texttt{deepfree}(x); \texttt{true} : \mathit{bool}}\ (\texttt{L-free})}{\cdot \vdash_L \lambda x{:}\tau.\texttt{deepfree}(x); \texttt{true} : \tau \multimap \mathit{bool}}\ (\texttt{L-fun})$$

Hence, it is possible to construct a linear language with the same expressive power as the intuitionistic language. Nevertheless, because linear types disallow sharing, linear programs have the potential to use exponentially more space than the corresponding intuitionistic program.

2.2 Controlling Space Reuse

The linear type system described in the previous section has the attractive feature that all memory management is achieved without the need for a tracing garbage collector. However, a programmer or compiler still has almost no control over when and how memory management is performed. As soon as the components of a pair are projected, the linear run-time system takes over and deallocates the heap binding associated with that pair. A programmer cannot make decisions on how to reuse the memory for this pair. If space reuse operations were explicit and under control of the application then a number of program optimizations, such as loop-invariant removal, common subexpression elimination, and cancelling pairs of inverse operations, would be possible.

For example, assume we want to write a simple swap function that inverts the order of the elements of a pair:

$$\lambda x{:}\mathit{bool} \otimes \mathit{bool}.\texttt{let}\, y, z = x\, \texttt{in}\, \langle z, y\rangle$$

Operationally, this function projects y and z from the pair x, deallocates x (which in practice would involve invoking memory manager routines that update the global data structures that keep track of free data), allocates another pair (invoking other memory management routines) and finally writes the results z and y into the two components of the freshly allocated pair. This is a lot of work for a simple function; our application would clearly be more efficient if the middle two steps were simply omitted. More reasonable code eschews the middle two operations and uses the component-wise projections $\texttt{let}\, y = x.i\, \texttt{in}\, e$ and imperative component assignments $x.i{:=}y$:

$$
\begin{array}{l}
\lambda x{:}bool \otimes bool. \\
\quad \texttt{let}\, y = x.1 \,\texttt{in} \\
\quad \texttt{let}\, z = x.2 \,\texttt{in} \\
\quad x.1{:=}z; \\
\quad x.2{:=}y; \\
\quad x
\end{array}
$$

Intuitively, this code is still be safe since it does not introduce aliasing, nor does it discard memory without recycling it. Moreover, it appears impossible to reconcile the five uses of x with a linear typing disciple. Nevertheless, in this section, I will show how to reorganize the linear type system in a simple way to make it possible to prove the safety of the code above.

2.2.1 Explicit Allocation, Initialization and Reuse

The first step towards our goal is to introduce a new type *junk* for unuseable objects. Using the junk type, we can remove the privileges for a data structure one at a time, instead of all at once, as we did in the previous section. Now, rather than using the pair-wise projection function $\texttt{let}\, x, y = e_1 \,\texttt{in}\, e_2$, one may project one component at a time using the expression $\texttt{let}\, z, x = e_1.i \,\texttt{in}\, e_2$, which binds x to the ith component of the pair e_1 and binds z to the pair itself. The typing rule for the component-wise projection gives the ith component the type junk to prevent that component from being used again in the future. Once all components of a pair have been projected, the memory structure may be deallocated using a free operation `free`. Unlike its cousin `deepfree`, `free` deletes only the top-level memory cell; it does not recursively delete all subcomponents.

For example, to implement the pair-wise projection exactly, using the new operations, one might write

$$
\begin{array}{llll}
\texttt{let}\, z_0 & = e_1 & \texttt{in} & \%\quad z_0{:}\tau_1 \otimes \tau_2 \\
\texttt{let}\, z_1, x & = z_0.1 & \texttt{in} & \%\quad z_1{:}junk \otimes \tau_2, x{:}\tau_1 \\
\texttt{let}\, z_2, y & = z_1.2 & \texttt{in} & \%\quad z_2{:}junk \otimes junk, x{:}\tau_1, y{:}\tau_2 \\
\texttt{free}(z_2); & & & \\
e_2 & & &
\end{array}
$$

At each step in the computation, some of the privileges of the pair are removed. However, the code is compatible with the type linear type system defined earlier. There is no need for contraction or weakening rules as each variable is used exactly once. Formally, the typing rules for these operations are straightfoward to define:

$$
\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \otimes \tau_2 \quad \Gamma_2, z{:}junk \otimes \tau_2, x{:}\tau_1 \vdash_L e_2 : \tau}{\Gamma_2, \Gamma_1 \vdash_L \texttt{let}\, z, x = e_1.1 \,\texttt{in}\, e_2 : \tau} \left(\begin{array}{c} z, x \notin Dom(\Gamma_2) \\ \tau_1 \neq junk \end{array} \right)
$$

$$
\frac{\Gamma_1 \vdash_L e_1 : \tau_1 \otimes \tau_2 \quad \Gamma_2, z{:}\tau_1 \otimes junk, x{:}\tau_2 \vdash_L e_2 : \tau}{\Gamma_2, \Gamma_1 \vdash_L \texttt{let}\, z, x = e_1.2 \,\texttt{in}\, e_2 : \tau} \left(\begin{array}{c} z, x \notin Dom(\Gamma_2) \\ \tau_2 \neq junk \end{array} \right)
$$

$$
\frac{\Gamma_1 \vdash_L e_1 : junk \otimes junk \quad \Gamma_2 \vdash_L e_2 : \tau}{\Gamma_2, \Gamma_1 \vdash_L \texttt{free}(e_1); e_2 : \tau}
$$

As in many other rules of the linear type system, the context is split into two disjoint parts and each part is used separately to check expressions e_1 and e_2. As a matter of convenience, I disallow projection of the junk value. If one of the components of a tuple is junk, it is perfectly safe to project that value, move it around and store it in another data structure, so this rule could be relaxed. However, none of these operations are particularly useful. On the other hand, the rule for `free` requires the two components of the pair to be junk and this restriction is essential. If the two components could be any type, say pairs themselves, then because the free operation is shallow, the component data structures would become unreachable and could not be collected.

For the next step in the development of our language, observe how the first projection rule specializes when e_1 is a simple variable (z):

$$\frac{\dfrac{}{z{:}\tau_1 \otimes \tau_2 \vdash_L z : \tau_1 \otimes \tau_2}\,(\texttt{LO-var}) \quad \Gamma_2, z'{:}\mathit{junk} \otimes \tau_2, x{:}\tau_1 \vdash_L e_2 : \tau}{\Gamma_2, z{:}\tau_1 \otimes \tau_2 \vdash_L \texttt{let}\, z', x = z.1 \,\texttt{in}\, e_2 : \tau}$$

Now, since the variable z is consumed when checking the pair, it cannot later appear free in e_2. Therefore, instead of binding a new name z' for the pair resulting from the projection operation, we could reuse the name z as in the following example:

$$\frac{\dfrac{}{z{:}\tau_1 \otimes \tau_2 \vdash_L z : \tau_1 \otimes \tau_2}\,(\texttt{LO-var}) \quad \Gamma_2, z'{:}\mathit{junk} \otimes \tau_2, x{:}\tau_1 \vdash_L e_2 : \tau}{\Gamma_2, z{:}\tau_1 \otimes \tau_2 \vdash_L \texttt{let}\, z, x = z.1 \,\texttt{in}\, e_2 : \tau}$$

In fact, if we consistently use the convention that we only project off variables and we always use the same variable name after the projection, the syntax for projection simplifies to $\texttt{let}\, x = z.i \,\texttt{in}\, e$. Moreover, since the `LO-var` rule always discharges the assumption on z, the typing rules for projections simplify as follows.

$$\frac{\Gamma, z{:}\mathit{junk} \otimes \tau_2, x{:}\tau_1 \vdash_L e : \tau}{\Gamma, z{:}\tau_1 \otimes \tau_2 \vdash_L \texttt{let}\, x = z.1 \,\texttt{in}\, e : \tau} \left(\begin{array}{c} x \notin \mathit{Dom}(\Gamma) \cup \{z\} \\ \tau_1 \neq \mathit{junk} \end{array} \right)$$

$$\frac{\Gamma, z{:}\tau_1 \otimes \mathit{junk}, x{:}\tau_2 \vdash_L e : \tau}{\Gamma, z{:}\tau_1 \otimes \tau_2 \vdash_L \texttt{let}\, x = z.2 \,\texttt{in}\, e : \tau} \left(\begin{array}{c} x \notin \mathit{Dom}(\Gamma) \cup \{z\} \\ \tau_2 \neq \mathit{junk} \end{array} \right)$$

By restricting `free` to operation on a variable, its typing rule can also be rephrased.

$$\frac{\Gamma \vdash_L e_2 : \tau}{\Gamma, z{:}\mathit{junk} \otimes \mathit{junk} \vdash_L \texttt{free}(z); e_2 : \tau}$$

Using a similar development, we can also decompose the rule for pair introduction into the more primitive expressions $\texttt{let}\, x = \texttt{new}() \,\texttt{in}\, e$, which allocates a pair and binds the result to x in e and $x.i{:=}y; e$, which assigns y to the ith component of x and continues with the expression e. In order for this operation to be safe, I require that the destination component of the assignment contain junk beforehand. If the component contained a proper data structure, then assignment would make that data structure unreachable garbage. The following typing rules specify the static semantics of `new` and assignment.

$$\frac{\Gamma, x{:}\mathit{junk} \otimes \mathit{junk} \vdash_L e : \tau}{\Gamma \vdash_L \texttt{let}\, x = \texttt{new}() \,\texttt{in}\, e : \tau} \; (x \notin \mathit{Dom}(\Gamma))$$

$$\frac{\Gamma, x{:}\tau_1 \otimes \tau_2 \vdash_L e : \tau}{\Gamma, x{:}\mathit{junk} \otimes \tau_2, y{:}\tau_1 \vdash_L x.1{:=}y; e : \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \otimes \tau_2 \vdash_L e : \tau}{\Gamma, x{:}\tau_1 \otimes \mathit{junk}, y{:}\tau_2 \vdash_L x.2{:=}y; e : \tau}$$

Suddenly, although the essential structure of the type system has changed little, we have exposed the operations for data construction and deconstruction, and have given our high-level, functional language a much lower-level, imperative feel. Furthermore, programmers have considerably more control over space reuse than in the high-level language. It is now easy to write the `swap` function mentioned earlier in an efficient imperative style:

```
λx:τ1 ⊗ τ2.
  let y = x.1 in    %  x:junk ⊗ τ2, y:τ1
  let z = x.2 in    %  x:junk ⊗ junk, y:τ1, z:τ2
  x.1:=z;           %  x:τ2 ⊗ junk, y:τ1
  x.2:=y;           %  x:τ2 ⊗ τ1
  x
```

2.2.2 Continuation-Passing Style

The programming constructs developed in the previous subsection rely on the invariant that all data structures be named using some variable. After each projection or assignment operation, the privileges associated with a name (as summarized by a type) are modified. For instance, after projecting the first component of a pair named x with type $\tau_1 \otimes \tau_2$, the privilege to use that component again is withdrawn and x is associated with type $\mathit{junk} \otimes \tau_2$ in the following code. Hence, propagation and processing of names is a central component of this programming style.

Some programming language constructs, particularly branching constructs, complicate the process of threading variable names through program. At the join point (*i.e.* the program point immediately after the branching construct) the variables from each branch must be unified. For instance, consider the following if statement:

```
if b (
  let x = new() in
  let y = new() in
  ...
|
  let z = new() in
  let w = new() in
  ...)
% use x,y or z,w ...
```

Assuming the code executed after the if statement needs to use the data structures allocated in the if statement then there must be some way to unify the names x and y with z and w so the following code can refer to the pairs stored there. There are several ways to accomplish this task. For instance, expressions could be allowed to return multiple results as in the following code:

```
let x1, x2 =
  if b (
    let x = new() in
    let y = new() in
    ...
    x, y
  |
    let z = new() in
    let w = new() in
    ...
    z, w)
% use names x1 and x2 ···
```

A second alternative, and the one I will adopt, is to move to a *continuation-passing style* (CPS) language [Fis72, Plo75]. In a CPS language, control never directly "returns" from a nested expression. Instead, when a nested expression completes its subcomputation, it invokes another function, called a continuation, passing the returned objects as arguments to the continuation. In this setting, the principle advantage to using CPS over a conventional *direct-style* language is that CPS is slightly simpler from a technical standpoint because all control-flow transfers occur via one mechanism (function call) as opposed to two mechanisms (function call and return). In a CPS language, one need only consider multiple parameters and not multiple results. In later chapters, when I consider polymorphic languages, CPS conveys a similar advantage at the level of types; I need only consider types as parameters to functions and need not consider them as results. A second benefit to CPS is that it makes the space required to store local variables across a function call explicit. The continuation's closure captures these local variables, and, like other closures, it is allocated when the continuation is defined and deallocated when the continuation is invoked.

The CPS variant of the code above follows.

```
if b then
  let x = new() in
  let y = new() in
  ...
  cont(x, y)
else
  let z = new() in
  let w = new() in
  ...
  cont(z, w)

% where cont = λ(x1:τ1, x2:τ2).··· (uses x1, x2)
```

Here *cont* is a continuation function taking two arguments x and y or z and w. Technically, *cont* is handled like any other (multi-argument) function. At run time, the names of the arguments (x and y, or z and w) are substituted for the formal parameters (x_1 and x_2) as in the standard interpretation of function call.

types	τ	$::=$	$junk \mid bool \mid \tau_1 \otimes \tau_2 \mid (\tau_1, \ldots, \tau_n) \multimap \mathbf{0}$
expressions	e	$::=$	$\texttt{let}\ x = b\ \texttt{in}\ e \mid \texttt{if}\ x\ (e_1 \mid e_2) \mid$ $\texttt{let}\ x = \texttt{new}()\ \texttt{in}\ e \mid \texttt{free}(x); e \mid$ $\texttt{let}\ x = y.i\ \texttt{in}\ e \mid x.i{:=}y; e \mid$ $\texttt{let}\ x = \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e_1\ \texttt{in}\ e_2 \mid x(x_1, \ldots, x_n) \mid$ $\texttt{halt}\ x$
pointers	p	$::=$	$_ \mid x$
storable values	h	$::=$	$b \mid \langle p_1, p_2 \rangle \mid \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
stores	S	$::=$	$\{x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\}$
programs	$\mathtt{P}$	$::=$	(S, e)

Figure 2.5: A Linear CPS Language: Syntax

CPS Syntax Figure 2.5 summarizes the syntax of a linear CPS language with explicit space reuse for pairs. The types include a type for junk, booleans and pairs. Function types have the form $(\tau_1, \ldots, \tau_n) \multimap \mathbf{0}$. The notation "$\multimap \mathbf{0}$" ($\mathbf{0}$ is pronounced "void") reflects the fact that CPS functions do not return. Most functions types have the form $(\tau_1, (\tau_2) \multimap \mathbf{0}) \multimap \mathbf{0}$. In other words, they take two arguments: an input that has type τ_1 and a continuation that has type $(\tau_2) \multimap \mathbf{0}$. The continuation is invoked when the corresponding direct-style function would have returned and, consequently, may be thought of as a return address. Intuitively, τ_2 can be thought of as a result type.

Most CPS expressions specify some primitive operation, such as a projection or allocation operation, which is followed by another expression. There are only two ways to terminate a sequence of expressions: a function call or the `halt` instruction. The function call is standard. The `halt` instruction is used to stop program execution. Its argument may be considered the result of the computation.

As before, we distinguish a class of storable values. The only difference is that during evaluation, when one of the components of a pair is used, it is marked as junk. Therefore, pair components may either be valid locations (variables) or junk (denoted using an underscore). Stores continue to be finite maps from variables to values and programs pair a store with an instruction stream.

CPS Operational Semantics The operational semantics for the CPS language is conceptually simpler than the operational semantics for the direct-style linear language. Because the beta redex or primitive operation that will be executed next never appears nested deep within a CPS expression, it is unnecessary to define evaluation contexts. Figure 2.6 presents the operational semantics formally.

The interesting rules involve pair construction and destruction. In particular, unlike the previous semantics, the projection operations (rules `cps0-p1` and `cps0-p2`) do not automatically deallocate the pair. Instead, when a component is projected, the projection operation marks that component as junk. Once a component is marked, it cannot be used, unless a new object is stored there. Of course, on a real machine, it may be undesirable to physically mark pair components after every projection. Fortunately, the marking process is not actually needed

$(S, \texttt{let}\, x = b\, \texttt{in}\, e) \longmapsto_{cps} (S\{x \mapsto b\}, e)$ where $x \notin Dom(S)$ (cps0-bool)

$(S\{x \mapsto \texttt{true}\}, \texttt{if}\, x\, (e_1 \mid e_2)) \longmapsto_{cps} (S, e_1)$ (cps0-ift)

$(S\{x \mapsto \texttt{false}\}, \texttt{if}\, x\, (e_1 \mid e_2)) \longmapsto_{cps} (S, e_2)$ (cps0-iff)

$(S, \texttt{let}\, x = \texttt{new}()\, \texttt{in}\, e) \longmapsto_{cps} (S\{x \mapsto \langle _, _ \rangle\}, e)$ where $x \notin Dom(S)$ (cps0-new)

$(S\{x \mapsto \langle p_1, p_2 \rangle\}, \texttt{let}\, y = x.1\, \texttt{in}\, e) \longmapsto_{cps} (S\{x \mapsto \langle _, p_2 \rangle\}, e[p_1/y])$ (cps0-p1)

$(S\{x \mapsto \langle p_1, p_2 \rangle\}, \texttt{let}\, y = x.2\, \texttt{in}\, e) \longmapsto_{cps} (S\{x \mapsto \langle p_1, _ \rangle\}, e[p_2/y])$ (cps0-p2)

$(S\{x \mapsto \langle p_1, p_2 \rangle\}, x.1{:=}y; e) \longmapsto_{cps} (S\{x \mapsto \langle y, p_2 \rangle\}, e)$ (cps0-a1)

$(S\{x \mapsto \langle p_1, p_2 \rangle\}, x.2{:=}y; e) \longmapsto_{cps} (S\{x \mapsto \langle p_1, y \rangle\}, e)$ (cps0-a2)

$(S\{x \mapsto \langle _, _ \rangle\}, \texttt{free}(x); e) \longmapsto_{cps} (S, e)$ (cps0-free)

$(S, \texttt{let}\, x = \lambda(y_1{:}\tau_1, \ldots, y_n{:}\tau_n).e_1\, \texttt{in}\, e_2) \longmapsto_{cps} (S\{x \mapsto \lambda(y_1{:}\tau_1, \ldots, y_n{:}\tau_n).e_1\}, e_2)$ where $x \notin Dom(S)$ (cps0-fun)

$(S\{x \mapsto \lambda(y_1{:}\tau_1, \ldots, y_n{:}\tau_n).e\}, x\, (x_1, \ldots, x_n)) \longmapsto_{cps} (S, e[x_1, \ldots, x_n / y_1, \ldots, y_n])$ (cps0-app)

Figure 2.6: A Linear CPS Language: Operational Semantics

for safe execution of the machine and it is easy enough to prove that, on well-typed programs, an abstract machine that does no marking simulates the semantics I have given here. The principle utility of the marking semantics is to simplify the proof of type soundness. By replacing pointers with junk, it becomes readily apparent that the anti-aliasing condition on the store, as formalized in the store typing rules, is preserved after each step in execution.

CPS Static Semantics Figures 2.7 and 2.8 summarizes the static semantics of CPS expressions. These rules are derived directly from the discussion in section 2.2.1. The rules for typing values (see figure 2.9) also need little explanation as they are almost identical to the rules for the direct-style linear lambda calculus. The program and store typing rules are actually identical to the rules for the linear lambda calculus.

Once again, it is straightforward to prove a simple type safety result.

Proposition 10 (Linear CPS Type Soundness) *If* $\vdash_{cps} (S, e)$ *and* $(S, e) \longmapsto^*_{cps} (S', e')$ *then* (S', e') *is not stuck.*

$\boxed{\Gamma \vdash_{cps} e}$

$$\frac{\Gamma, x{:}bool \vdash_{cps} e}{\Gamma \vdash_{cps} \texttt{let}\, x = b \,\texttt{in}\, e} \ (x \notin Dom(\Gamma)) \ (\texttt{cps-bool})$$

$$\frac{\Gamma \vdash_{cps} e_1 \quad \Gamma \vdash_{cps} e_2}{\Gamma, x{:}bool \vdash_{cps} \texttt{if}\ x\ (e_1 \mid e_2)} \ (\texttt{cps-if})$$

$$\frac{\Gamma, x{:}junk \otimes junk \vdash_{cps} e}{\Gamma \vdash_{cps} \texttt{let}\, x = \texttt{new}() \,\texttt{in}\, e} \ (x \notin Dom(\Gamma)) \ (\texttt{cps-new})$$

$$\frac{\Gamma, x{:}\tau_1, y{:}junk \otimes \tau_2 \vdash_{cps} e}{\Gamma, y{:}\tau_1 \otimes \tau_2 \vdash_{cps} \texttt{let}\, x = y.1 \,\texttt{in}\, e} \ \left(\begin{array}{c} x \notin Dom(\Gamma) \cup \{y\} \\ \tau_1 \neq junk \end{array}\right) \ (\texttt{cps-p1})$$

$$\frac{\Gamma, x{:}\tau_2, y{:}\tau_1 \otimes junk \vdash_{cps} e}{\Gamma, y{:}\tau_1 \otimes \tau_2 \vdash_{cps} \texttt{let}\, x = y.2 \,\texttt{in}\, e} \ \left(\begin{array}{c} x \notin Dom(\Gamma) \cup \{y\} \\ \tau_2 \neq junk \end{array}\right) \ (\texttt{cps-p2})$$

$$\frac{\Gamma, x{:}\tau_1 \otimes \tau_2 \vdash_{cps} e}{\Gamma, x{:}junk \otimes \tau_2, y{:}\tau_1 \vdash_{cps} x.1{:=}y; e} \ (\texttt{cps-a1})$$

$$\frac{\Gamma, x{:}\tau_1 \otimes \tau_2 \vdash_{cps} e}{\Gamma, x{:}\tau_1 \otimes junk, y{:}\tau_2 \vdash_{cps} x.2{:=}y; e} \ (\texttt{cps-a2})$$

$$\frac{\Gamma \vdash_{cps} e}{\Gamma, x{:}junk \otimes junk \vdash_{cps} \texttt{free}(x); e} \ (\texttt{cps-free})$$

$$\frac{\Gamma_1, x_2{:}\tau_2, x_1{:}\tau_1, \Gamma_2 \vdash_{cps} e}{\Gamma_1, x_1{:}\tau_1, x_2{:}\tau_2, \Gamma_2 \vdash_{cps} e} \ (\texttt{cps-exchange})$$

Figure 2.7: A Linear CPS Language: Static Semantics, Expressions

$\boxed{\Gamma \vdash_{cps} e}$

$$\frac{\Gamma_1 \vdash_{cps} \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e_1 : (\tau_1, \ldots, \tau_n) \multimap \mathbf{0} \qquad \Gamma_2, x{:}(\tau_1, \ldots, \tau_n) \multimap \mathbf{0} \vdash_{cps} e_2}{\Gamma_1, \Gamma_2 \vdash_{cps} \mathtt{let}\, x = \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e_1 \,\mathtt{in}\, e_2} \; (x \notin Dom(\Gamma_2)) \; (\texttt{cps-fun})$$

$$\frac{}{x{:}(\tau_1, \ldots, \tau_n) \multimap \mathbf{0}, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash_{cps} x(x_1, \ldots, x_n)} \; (\texttt{cps-app})$$

$$\frac{}{x{:}bool \vdash_{cps} \mathtt{halt}\; x} \; (\texttt{cps-halt})$$

Figure 2.8: A Linear CPS Language: Static Semantics, expressions Cont.

$\boxed{\Gamma \vdash_{cps} p : \tau}$

$$\frac{}{x{:}\tau \vdash_{cps} x : \tau} \; (\texttt{cps-var})$$

$$\frac{}{\cdot \vdash_{cps} _ : junk} \; (\texttt{cps-junk})$$

$\boxed{\Gamma \vdash_{cps} v : \tau}$

$$\frac{}{\cdot \vdash_{cps} b : bool} \; (\texttt{cps-bool})$$

$$\frac{\Gamma_1 \vdash_{cps} p_1 : \tau_1 \qquad \Gamma_2 \vdash_{cps} p_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash_{cps} \langle p_1, p_2 \rangle : \tau_1 \otimes \tau_2} \; (\texttt{cps-pair})$$

$$\frac{\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash_{cps} e}{\Gamma \vdash_{cps} \lambda(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : (\tau_1, \ldots, \tau_n) \multimap \mathbf{0}} \; (x_1, \ldots, x_n \notin Dom(\Gamma)) \; (\texttt{cps-fun})$$

Figure 2.9: A Linear CPS Language: Static Semantics, Values

CPS Conversion The continuation-passing style transformation was developed in the 70s by Fischer[Fis72] and Plotkin [Plo75]. Since then, many facets of the translation (both typed and untyped variants) have been studied in great detail [DF92, SF93, HL93, DDP99]. I have nothing further to add on this topic at this point but the interested reader may wish to refresh his or her memory and I highly recommend the splendid exposition by Danvy and Filinski [DF92].

2.3 Discussion

The continuation-passing style transformation is the first phase in several successful compilers for functional programming languages including the Rabbit [Ste78], Orbit [KKR+86] and Standard ML of New Jersey [AM91] compilers. CPS conversion is also the first stage in a theoretical, type-preserving compiler that maps the polymorphic lambda calculus into a Typed Assembly Language [MWCG99, MWCG98].

The next major step in these compilers is closure conversion. This transformation replaces every function with a pair of a pointer to closed, top-level code and an *environment* for that code, which is a tuple containing the values of the (formerly) free variables of the nested function. After closure conversion, the problem of compiling an intermediate language is no harder than compiling a language like C that contains first-class code pointers but no nested functions.

A closed, continuation-passing style intermediate language gives great control over memory management. Activation records (stack frames) are represented explicitly as continuation closures and closures of all types are represented as simple memory blocks (pairs or, more generally, tuples). Hence, the techniques discussed earlier in this chapter can be used to manage all the data structures necessitated by functions. More specifically, allocation, deallocation and reuse of stack frames and of closures can be managed explicitly by the application program and closures are explicit in the language. In fact, if lists were added to the language, applications could manage all their own memory; primitives such as `new` and `free` would be unnecessary. Every time a pair is required, it could be removed from a free list under programmer control and whenever a pair is unneeded it could be returned to the free list, just as Baker suggests in his article describing an (untyped) "lively linear LISP" [Bak92].

The primary advantage of a linearly typed intermediate language that controls all its own memory resources is that it would have a tiny trusted computing base: Only a simple interpreter, the underlying hardware and the type checker need to be trusted, not the garbage collector. Such a language would make an attractive choice as platform for verifying untrusted mobile code. Alternatively, linear types could be smoothly integrated with existing research on Typed Assembly Language [MWCG99, MCGW98] or, with a little more work, other frameworks for machine-level proof-carrying code [Nec97, Koz98]. By verifying machine code directly, the interpreter can be eliminated from the trusted computing base, leaving only the machine and type checker. The complex invariants linking a garbage collector to application code would not need to be trusted.

Despite the memory-management opportunities afforded by working in a low-level language with explicit closures, I will continue to develop high-level languages with implicit closures. In general, closure-converted code (particularly typed, closure-converted code) is quite complicated and it would obscure the central technical work. When the details of closure implementation are relevant to the discussion, I will point them out.

2.3.1 Related Work

The elegance, simplicity and wealth of applications of Girard's linear logic has lead to extensive research in the logic, programming languages and semantics communities. For a superb introduction to linear logic and the relationship with intuitionistic logic, see Wadler [Wad93]. The volume of research related to linear logic makes it impossible to mention it all, so I will restrict myself to only the most relevant work, which involves operational interpretations of linear logic, optimization and applications in functional programming languages

Lafont [Laf88] was one of the first to study an operational interpretation of the linear lambda calculus in detail. He defines a linear abstract machine, which is derived from the Categorical Abstract Machine [GC85]. The linear abstract machine is not typed itself, but it can interpret a linear lambda calculus and performs its own storage management.

Abramsky [Abr93] studied both intuitionistic and classical linear logic and gives them operational interpretations. His interpretation of intuitionistic linear logic leads to a linear lambda calculus whereas his interpretation of classical linear logic leads to a concurrent programming model. Lincoln and Mitchell [LM92] develop a semantics that is quite closely related to Abramsky's intuitionistic interpretation and they also analyze type inference techniques for linear lambda calculi.

In some of these early operational models, there was considerable confusion about their precise storage management properties. Unlike the *purely linear* language presented in this chapter, Lafont, Abramsky, and Lincoln and Mitchell's languages are referred to as *intuitionistic linear* because of the presence of the intuitionistic types $!\tau$ in addition to the linear types $\tau \otimes \tau$ and $\tau \multimap \tau$. Objects of linear type are never copied (or discarded) *directly*, only the intuitionistic objects are. However, the precise definition of the copy operation on *intuitionistic* objects has considerable impact on the memory management properties of the *linear* fragment of the language. More specifically, unless a value of intuitionistic type is completely recomputed each time it is used or else the intuitionistic copy operation actually performs a deep copy then programs may evaluate to states where objects of linear type have multiple pointers to them! This surprising property was observed by several research groups including Wadler [Wad90], Lincoln and Mitchell [LM92], and Chirimar, Gunter and Riecke [CGR92].

In order to avoid the expense of a deep copy and yet collect linear objects, Chirimar, Gunter and Riecke [CGR92, CGR96] prove that an intuitionistic linear lambda calculus can be correctly implemented using reference counting. One of the main advantages of their operational model over previous models is the use of a store that maps locations to values. This construction makes sharing explicit and allows them to reason correctly about reference counts. Launchbury's semantics for lazy languages [Lau93] and Morrisett, Felleisen and Harper's investigation of the semantics of garbage collection [MFH95, MH97] use related ideas to analyze sharing. Previous work on linear type systems used more abstract storage models that obscured the semantics of memory management.

Inspired by Chirimar *et al.*, Turner and Wadler [TW99] decided to further analyze the difference between the semantics that recomputes intuitionistic values each time they are used and the semantics that shares intuitionistic values. Like Chirimar *et al.*, they use a store to make aliasing explicit. Turner and Wadler then define two sets of reduction rules and formally prove that the rules that recompute *intuitionistic* values ensure every *linear* value has exactly one pointer to it and the rules that allow *intuitionistic* values to share do not ensure every *linear* value has exactly one pointer to it. The main problem with objects of type $!\tau$ is that they may contain subcomponents of linear type, which when extracted multiple times but not copied, become shared.

In terms of storage management, Turner and Wadler's formal analysis puts us in the extremely uncomfortable position of having to choose between making copies of intuitionistic values or allowing values of linear type to be shared. In my opinion, this analysis indicates that the ! operator, when used without restriction, really is not the right operator for a practical memory managment system.[4] Wadler's earlier work on linear type systems [Wad90] may be a better approach. Instead of using a single intuitionistic type constructor ($!\tau$) and a replication construct ($!e$), Wadler used two distinct sets of types and terms: a set of intuitionistic types ($\rightarrow$, $\times$, etc.) and terms, and a set of linear types ($\multimap$, $\otimes$, etc.) and terms. He disallowed linear types from appearing inside intuitionistic ones, thereby ensuring that intuitionistic types share and linear types do not.

The semantics most closely related to the one given in this chapter is Turner and Wadler's recomputing semantics. Like them, I use an explicit store so it is possible to express the difference between objects with multiple pointers to them and objects with a single pointer to them. On the other hand, unlike Turner and Wadler, I do not use the intuitionistic types $!\tau$. Instead, I allow linear objects to be explicitly copied. This decision makes it possible to encode an intuitionistic calculus, but it diverges from the traditional logical interpretation of linearity, which views linear objects as "uncopyable." The essence of an object of linear type, as I interpret it, is the property of being "unshared."

This sharing interpretation implies a close relationship with Reynold's syntactic control of interference (SCI) [Rey78, Rey89, O'H93, O'H00]. The goal of SCI is to prevent aliasing so that assignment to one program identifier does not influence (*i.e.* interfere with) the values referred to by any other program identifier. As in linear logic, there is close control over the use of the contraction rule. O'Hearn [O'H00] elegantly explains further similarities and dissimilarities between the two systems.

There are a number of other type systems based on notions of linearity that have been applied to optimization problems or to the problem of controlling effects in purely functional languages. For instance, Guzman and Hudak [GH90] developed the single-threaded polymorphic lambda calculus in order to allow in-place update of data structures in a purely functional languages such as Haskell [PH99]. Wadler [Wad91] has studied a system of *use types* to accomplish similar ends. Turner, Wadler and Mossin [TWM95] and later Wansbrough and Peyton Jones [WP99] have discovered analysis techniques based on linear typing for avoiding thunk update and for making decisions about inlining and other program transformations. Barendsen and Smetsers [BS93] have developed *uniqueness types* that are used for inplace update as well as structuring I/O and concurrency operations in the purely functional language Clean [AP95].

2.3.2 Limitations

Garbage collection based on the idea of linear typing is certainly not without its limitations. Indeed, linear typing can negatively impact both the time and the maximum amount of space required to run a program. For example, a linear tree may require exponentially more space than a dag that shares internal nodes to represent the same information. Likewise, graphs, general recursive functions, and any data structure that contains cycles cannot be represented without finding a way to break the cycle. Finally, a variety of standard data structures used in compilers such as static links or displays [ASU86] and some implementations of exceptions create shared data structures.

In the next several chapters of this thesis, I will develop more expressive languages that can represent shared data structures, yet safely and explicitly reuse memory.

[4] Unless Chirimar *et al.*'s reference counting semantics is used.

Chapter 3

Aliasing

Of the terrible doubt of appearances,
Of the uncertainty after all—that we may be deluded,
That may-be reliance and hope are but speculations after all ...

– *Walt Whitman. Of the Terrible Doubt of Appearances.*

The linear type $\tau_1 \otimes \tau_2$ captures an extremely valuable memory management invariant: There is only one access path to any value with this type. If x has type $\tau_1 \otimes \tau_2$ then the only way to access x's data is through x itself. Therefore, x may be safely reused to store objects of different types and deallocation does not leave potentially dangerous dangling pointers around. Unfortunately, as discussed in the previous chapter, the restriction to a single access path makes it impossible to construct a number of important data structures and to implement natural, efficient algorithms. The goal of this chapter is to introduce a new language of *alias types* that makes it possible to represent shared data structures and yet retain the capacity to safely reuse or deallocate memory. The material in this chapter is derived from research I have done jointly with Greg Morrisett and Fred Smith [SWM00, WM00].

3.1 Informal Development

Chapter two suggested that one of the keys to meta-level reasoning about sharing is to explicitly represent stored values as two-part structures consisting of a block of memory and a location (the memory's address). The central idea of this chapter is to show how to reflect this two-part structure into a type system so that a mechanical type checker can reason about sharing within the language itself.

Locations One difficulty with separating pointer types from memory types is that in order to use a pointer to update or dereference memory, there must be some way to relate the pointer to the memory block it points to. In other words, the type system must contain some mechanism for expressing the *dependency* between pointers and memory. The traditional way to express such dependencies is to allow types to include expressions. For example, the type for memory blocks might include pointer expressions to indicate which pointers point to the block. In this design, the algorithms for deciding the equivalence of two types (a fundamental procedure in any type checker) will require a decision procedure for the equivalence of expressions. Therefore, if we allow arbitrary expressions to appear inside types, type checking will be as hard as determining when two lambda calculus expressions are equal. We will quickly find ourselves

working in an undecidable system on par with Martin-Löf type theory [ML82], Nuprl [CAB+86], or the Calculus of Constructions [CH88]. In order to ensure the type system is decidable, we introduce a special syntactic class of memory locations ℓ that is distinct from the expressions. Only locations, not expressions, will be found in types. Since testing the equivalence of two locations is a simple syntactic operation, type checking will be relatively easy.[1]

Store Types In the alias type system, every heap-allocated data structure has two parts to its type, connected together using locations. A pair, for example, may be factored into:

- A type for the memory, called a *store type*, that takes the form $\{\ell \mapsto \langle\tau_1, \tau_2\rangle\}$. This type states that at location ℓ there exists a memory block containing components with types τ_1 and τ_2.
- A type for a pointer to the location: $ptr(\ell)$. This type is a *singleton type*—any pointer described by $ptr(\ell)$ is a pointer to the one location ℓ and to no other location.

Complex data structures can be described by joining together a number of capabilities using the $*$ constructor. For instance, an unshared pair of pairs of booleans might be described using the following type:

$$\{\ell_1 \mapsto \langle ptr(\ell_2), ptr(\ell_3)\rangle\} * \{\ell_2 \mapsto \langle bool, bool\rangle\} * \{\ell_3 \mapsto \langle bool, bool\rangle\}$$

For the sake of brevity, we often abbreviate the store type $\{\ell_1 \mapsto \tau_1\} * \cdots * \{\ell_n \mapsto \tau_n\}$ with $\{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}$. The meta-variable C ranges over store types.

A crucial aspect of this division is that store types, not pointers, bestow the privilege to access data structures. For example, suppose a variable x is a pointer and has type $ptr(\ell)$. In order to dereference x at a particular program point p, the type checker must be able to prove that the store has a type containing the constraint $\{\ell \mapsto \langle\tau_1, \ldots, \tau_n\rangle\}$ at p. Once this obligation has been satisfied, access will be granted. If the obligation cannot be satisfied, then the type checker will reject the program, *even though x is a well-formed pointer.* In this language, deallocation can leave dangling pointers behind. Dangling pointers are well-formed objects and may be safely stored in data structures or passed to functions, but they cannot be dereferenced. Since x may well be a dangling pointer, the type system conservatively rejects programs that access x without presenting evidence that the access is safe.

Because store types play a central role in controlling data access rights, they may be viewed as a form of *static capability* and I will sometimes refer to them as such. Like their dynamic counterparts, which are found in operating systems such as Hydra [WLH81] or the J-Kernel [HCC+98], object access involves a two-part protocol: The system first verifies that a client possesses the appropriate capability and then grants access. However, unlike their dynamic counterparts, static capabilities incur no run-time overhead — all memory management operations are safely checked at compile time.

[1] There are other ways to ensure the type system is decidable. For instance, instead of defining a distinct class of locations, I could use expression variables x and y to represent locations as I did in chapter two and allow expression variables (not general expressions) to appear in types. However, such a choice makes the type system slightly more complicated as type well-formedness would then depend upon value contexts Γ (which themselves contain types). The use of a syntactically separate class of locations is a convenient simplification with no loss in expressive power.

Allocation and Deallocation Allocation operations extend store types. For instance, if the store is described by C before an allocation operation, then after allocation the store may be described by the extended type $C * \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$ where ℓ is a fresh location. If we consider the capability metaphor again, then object allocation is the act of *granting* new capabilities. Operations for memory deallocation have the opposite effect. If $C * \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$ describes the current store, then after deallocating the object in location ℓ, the program is left with a store of type C. In other words, object deallocation *revokes* capabilities. Section 3.2 describes the static semantics for these operations in more formal detail.

Structural Rules for Store Types Pointers are cheap: They fit in machine registers and copying a pointer can be as inexpensive as a single move instruction. In contrast, a memory block is much more expensive to allocate, copy or deallocate. Each of these operations requires non-trivial work. Hence, while we can be somewhat care-free with pointers, copying and discarding them at will, we will have to be much more prudent with memory blocks. More to the point, I will treat pointers intuitionistically and apply a linear typing discipline to memory.

A linear store typing discipline naturally allows an exchange rule. Informally, the inference rule is admissible:

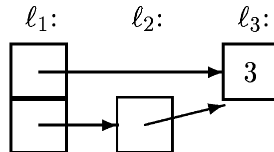
$$\frac{C_1 * \{\ell_2 \mapsto \tau_2\} * \{\ell_1 \mapsto \tau_1\} * C_2 \vdash_A e}{C_1 * \{\ell_1 \mapsto \tau_1\} * \{\ell_2 \mapsto \tau_2\} * C_2 \vdash_A e} \; (\texttt{A-exchange})$$

Also like the linear type system, contraction and weakening rules are not admissible:

$$\frac{C_1 * \{\ell \mapsto \tau\} * \{\ell \mapsto \tau\} * C_2 \vdash_A e}{C_1 * \{\ell \mapsto \tau\} * C_2 \vdash_A e} \; (\texttt{Wrong})$$

$$\frac{C_1 * C_2 \vdash_A e}{C_1 * \{\ell \mapsto \tau\} * C_2 \vdash_A e} \; (\texttt{Wrong})$$

Sample Store Types The principle reason the new type system is more expressive than the linear type system is that the pointer types $ptr(\ell)$ are intuitionistic. They may be discarded and copied without performing a deep copy of the underlying memory structure. Therefore multiple occurrences of the type $ptr(\ell)$ may appear in a store type; each occurence represents an alias. This flexibility makes it possible to construct and reason about shared structures. For example, a DAG:



may be represented using these constraints:

$$\{\ell_1 \mapsto \langle ptr(\ell_2), ptr(\ell_3) \rangle, \ell_2 \mapsto \langle ptr(\ell_3) \rangle, \ell_3 \mapsto \langle int \rangle\}$$

Notice that the type $ptr(\ell_3)$ appears twice in the store type. Whenever two objects are described by equal singleton types, then the objects themselves are identical. In the case of pointers, equal singleton types provide us with useful *must alias* information.

Function Types In the linear languages of the previous chapter, the shape of the store passed to a function was completely specified by the types of the standard function arguments. In the new language, the store may have much more complex aliasing structure. Therefore, every function must specify store shape in addition to specifying the types of its arguments. As in section 2.2, functions are defined in continuation-passing style. Hence, a typical function type has the form

$$(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}$$

where C types the store of the calling context and τ_1 through τ_n give types to the function arguments as usual. Unlike in chapter two, I will not attempt to control the space used by function closures here.[2] Consequently, I will treat functions intuitionistically (as I do data pointers) — they may be used multiple times or not all. To remind us of this fact, the syntax of function types uses the intuitionistic $\to$ rather than the linear $\multimap$.

With these points in mind, I might give a function `deref` that projects the value stored at location ℓ and leaves the store unaltered the type:

$$(\{\ell \mapsto \langle\tau\rangle\}, ptr(\ell), \tau_{cont}) \to \mathbf{0}$$

where $\tau_{cont} = (\{\ell \mapsto \langle\tau\rangle\}, \tau) \to \mathbf{0}$. In order to call `deref`, the current store must have the type $\{\ell \mapsto \langle\tau\rangle\}$. Upon "return," the continuation may assume the store continues to have type $\{\ell \mapsto \langle\tau\rangle\}$.

3.1.1 Abstraction Mechanisms

Any particular store can be represented exactly using the store types that have been described so far[3], even stores containing cyclic data structures. For example, a node containing a pointer to itself may be represented with the type $\{\ell \mapsto \langle ptr(\ell)\rangle\}$. However, the principal difficulty in describing aliasing relationships is not specifying one particular store but being able to specify a class of stores using a single compact representation. By specifying a wide class of stores as a function precondition, it is possible to reuse the function code in variety of different contexts. In the following paragraphs, I informally describe a number of type-theoretic abstraction mechanisms that can be used to describe classes of pointer-rich data structures. Section 3.2 describes the type structure formally.

Location Polymorphism The interpretation of a location ℓ is a specific machine address, perhaps the address `0xFF2644`. However, in general, the physical address of an object is inconsequential to the algorithm being executed. The relevant information is the connection between the location ℓ, the contents of the memory residing there, and the pointers $ptr(\ell)$ to that location. Routines that only operate on specific concrete locations are almost useless. For example, the `deref` function mentioned above can only operate on the single concrete location ℓ (address `0xFF2644`) and therefore we would have to implement a different dereference function to dereference another location (say address `0xFFAC00`). In fact, in general, we have to implement a different dereference function for every location in the store!

[2] I will add existentials to the language. Therefore, it is powerful enough to encode closure conversion in the style of Morrisett *et al.* [MWCG98].

[3] I cannot represent a store containing a pointer into the middle of a memory block.

By introducing *location polymorphism*, it is possible to abstract away from the concrete location ℓ using a variable location ζ, but retain the necessary dependencies. The location-polymorphic `deref` promotes greater code reuse than the monomorphic variant, but performs exactly the same operations. Its type is:

$$\forall[\zeta{:}\mathtt{Loc}].(\{\zeta \mapsto \langle\tau\rangle\}, ptr(\zeta), \tau_{cont}) \rightarrow \mathbf{0}$$

where $\tau_{cont} = (\{\zeta \mapsto \langle\tau\rangle\}, \tau) \rightarrow \mathbf{0}$. `Loc` is the *kind* of the variable ζ. Soon, I will be adding other kinds of type variables and, formally, I use kinding annotations to distinguish between them. However, normally, the kind of the variable is evident from context, and in any event, I will use the meta-variable ζ alone for variables of location kind. Therefore, in examples, I will normally omit kind annotations. I use the meta-variable η to refer to locations generically, either concrete (ℓ) or variable (ζ).

Store Polymorphism Any specific routine only operates over a portion of the store. In order to use that routine in multiple contexts, we abstract irrelevant portions of the store using *store polymorphism*. A store described by the constraints $\epsilon * \{\eta \mapsto \langle\tau\rangle\}$ contains some store of unknown size and shape ϵ as well as a location η containing objects with type τ.

The `deref` function is good example of function that should be made store polymorphic as well as location polymorphic. An even better type to give the `deref` function is:

$$\forall[\epsilon{:}\mathtt{Store}, \zeta{:}\mathtt{Loc}].(\epsilon * \{\zeta \mapsto \langle\tau\rangle\}, ptr(\zeta), \tau_{cont}) \rightarrow \mathbf{0}$$

where $\tau_{cont} = (\epsilon * \{\zeta \mapsto \langle\tau\rangle\}, \tau) \rightarrow \mathbf{0}$. The kind `Store` is the kind of store types.

Type Polymorphism The final degree of polymorphism is useful for compiling languages like ML or Haskell. *Type polymorphism* allows functions to accept arguments with a variety of different types. Using type, store and location polymorphism, the best type this language can give the `deref` function is

$$\forall[\alpha{:}\mathtt{Type}, \epsilon{:}\mathtt{Store}, \zeta{:}\mathtt{Loc}].(\epsilon * \{\zeta \mapsto \langle\alpha\rangle\}, ptr(\zeta), \tau_{cont}) \rightarrow \mathbf{0}$$

where $\tau_{cont} = (\epsilon * \{\zeta \mapsto \langle\alpha\rangle\}, \alpha) \rightarrow \mathbf{0}$.

In their most general form, functions types have the shape $\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}$ where Δ is a list of type variable-kind pairs. I will continue to write monomorphic types as $(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}$, but they should be considered abbreviations for a general function type with an empty type context (*i.e.* for $\forall[\cdot].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}$).

Unions Unlike polymorphic types, unions provide users with the abstraction of one of a finite number of choices. A memory block that holds either junk or a pointer may be encoded using the type $\langle junk\rangle \cup \langle ptr(\eta)\rangle$. However, in order to use the contents of the block safely, there must be some way to detect which element of the union the underlying value actually belongs to. There are several ways to perform this test: through a pointer equality test with an object of known type, by descriminating between small integers (including null/0) and pointers, or by distinguishing between components using explicit tags. All of these options will be useful in an implementation, but here I concentrate on the third option (see section 3.5.3 for more discussion of this point). Hence, the alternatives above will be encoded using the type $\langle \mathcal{S}(1), junk\rangle \cup \langle \mathcal{S}(2), ptr(\eta)\rangle$ where $\mathcal{S}(i)$ is another form of singleton type — the type containing only the integer i. As a second example, a store containing a single boolean value in location η has the shape $\{\eta \mapsto \langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2)\rangle\}$.

Recursion As yet, I have defined no mechanism for describing regular repeated structure in the store. I use standard recursive types of the form $\mu\alpha.\tau$ to capture this notion. However, recursion by itself is not enough. Consider an attempt to represent a store containing a linked list where the tag $\mathcal{S}(1)$ signals an empty list, and the tag $\mathcal{S}(2)$ signals a non-empty list:[4]

$$\{\eta \mapsto \mu\alpha.\langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2), \alpha\rangle\}$$

An unfolding of this definition results in the type

$$\{\eta \mapsto \langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2), \langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2), List\rangle\rangle\}$$

rather than the type

$$\{\eta \mapsto \langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2), ptr(\eta')\rangle, \eta' \mapsto \langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2), List\rangle\}$$

The former type describes a number of memory blocks flattened into the same location whereas the latter type describes a linked collection of disjoint nodes. If a linked list is what we are after, then the former type will not do.

Encapsulation In order to represent linked recursive structures properly, each unfolding must encapsulate its own portion of the store. I use an existential type for this purpose. Hence, a sensible representation for linked lists is

$$\mu\alpha.\langle\mathcal{S}(1)\rangle \cup \exists[\zeta{:}\mathtt{Loc} \mid \{\zeta \mapsto \alpha\}].\langle\mathcal{S}(2), ptr(\zeta)\rangle$$

The existential $\exists[\zeta{:}\mathtt{Loc} \mid \{\zeta \mapsto \tau_1\}].\tau_2$ may be read "there exists some location ζ, different from all others in the program, such that ζ contains an object of type τ_1, and the value contained in this data structure has type τ_2. More generally, an existential has the form $\exists[\Delta \mid C].\tau$. It abstracts a sequence of type variables with their kinds, Δ, and encapsulates a store described by some constraints C. Once again, in our examples, we will omit the kinds from the sequence Δ as they are clear from context. A similar definition gives rise to trees:

$$\mu\alpha.\langle\mathcal{S}(1)\rangle \cup \exists[\zeta_1, \zeta_2 \mid \{\zeta_1 \mapsto \alpha, \zeta_2 \mapsto \alpha\}].\langle\mathcal{S}(2), ptr(\zeta_1), ptr(\zeta_2)\rangle$$

Notice that the existential abstracts a pair of locations and that both locations are bound in the store. From this definition, it follows that the two subtrees are disjoint. For the sake of contrast, a DAG in which every node has a pair of pointers to a single successor is coded as follows. Here, reuse of the same location variable ζ indicates aliasing.

$$\mu\alpha.\langle\mathcal{S}(1)\rangle \cup \exists[\zeta \mid \{\zeta \mapsto \alpha\}].\langle\mathcal{S}(2), ptr(\zeta), ptr(\zeta)\rangle$$

Cyclic lists and trees with leaves that point back to their roots also cause little problem—simply replace the terminal node with a memory block containing a pointer type back to the roots.

$$CircularList = \{\zeta_1 \mapsto \mu\alpha.\langle\mathcal{S}(1), ptr(\zeta_1)\rangle \cup \exists[\zeta_2 \mid \{\zeta_2 \mapsto \alpha\}].\langle\mathcal{S}(2), ptr(\zeta_2)\rangle\}$$

$$CircularTree = \{\zeta_1 \mapsto \mu\alpha.\langle\mathcal{S}(1), ptr(\zeta_1)\rangle \cup \exists[\zeta_2, \zeta_3 \mid \{\zeta_2 \mapsto \alpha, \zeta_3 \mapsto \alpha\}].\langle\mathcal{S}(2), ptr(\zeta_2), ptr(\zeta_3)\rangle\}$$

[4]Throughout we use the convention that union binds tighter than the recursion operator.

Parameterized Recursive Types One common data structure we are unable to encode with the types described so far is the doubly-linked list. Recursive types only "unfold" in one direction, making it easy to represent pointers from a parent "down" to its children, or all the way back up to the top-level store, but much more difficult to represent pointers that point back up from children to their parents, which is the case for doubly-linked lists or trees with pointers back to their parent nodes. One solution to this problem is to use parameterized recursive types to pass a parent location down to its children. In general, a parameterized recursive type has the form $\mathtt{rec}\,\alpha\,(\beta_1{:}\kappa_1,\ldots,\beta_n{:}\kappa_n).\tau$ and has kind $(\kappa_1,\ldots,\kappa_n) \rightarrow \mathtt{Type}$. We will continue to use unparameterized recursive types $\mu\alpha.\tau$ in examples and consider them to be an abbreviation for $\mathtt{rec}\,\alpha\,().\tau[\alpha\,()/\alpha]$. Once again, kinds will be omitted when they are clear from the context. Trees in which each node has a pointer to its parent may be encoded as follows.

$$\begin{array}{l}\{\zeta_{root} \mapsto \langle \mathcal{S}(2), ptr(\zeta_L), ptr(\zeta_R)\rangle\} * \\ \{\zeta_L \mapsto REC\,(\zeta_{root}, \zeta_L)\} * \\ \{\zeta_R \mapsto REC\,(\zeta_{root}, \zeta_R)\}\end{array}$$

where

$$\begin{array}{l}REC = \\ \quad \mathtt{rec}\,\alpha\,(\zeta_{prt}, \zeta_{curr}). \\ \qquad \langle \mathcal{S}(1), ptr(\zeta_{prt})\rangle \cup \\ \qquad \exists[\zeta_L, \zeta_R \mid \{\zeta_L \mapsto \alpha\,(\zeta_{curr}, \zeta_L)\} * \\ \qquad\qquad\qquad \{\zeta_R \mapsto \alpha\,(\zeta_{curr}, \zeta_R)\}]. \\ \qquad\qquad\qquad\quad \langle \mathcal{S}(2), ptr(\zeta_L), ptr(\zeta_R), ptr(\zeta_{prt})\rangle\end{array}$$

The tree has a root node in location ζ_{root} that points to a pair of children in locations ζ_L and ζ_R, each of which are defined by the recursive type *REC*. *REC* has two arguments, one for the location of its immediate parent ζ_{prt} and one for the location of the current node ζ_{curr}. Either the current node is a leaf, in which case it points back to its immediate parent, or it is an interior node, in which case it contains pointers to its two children ζ_L and ζ_R as well as a pointer to its parent. The children are defined recursively by providing the location of the current node (ζ_{curr}) for the parent parameter and the location of the respective child (ζ_L or ζ_R) for the current pointer.

3.2 Formal Syntax and Semantics

In this section, I define the syntax and semantics of the language, starting with the type constructors in subsection 3.2.1 and following with the term constructs in subsection 3.2.2.

3.2.1 Type Constructors

Figure 3.1 presents the formal syntax for the type constructor language. I use the term *type constructor* (and meta-variable c) when speaking generically about locations, store types or "standard" types. The meta-variable β is used to range over constructor variables generically. The domain of a type context Δ ($Dom(\Delta)$) is the set of type constructor variables in the first-components of the list. If Δ is $\beta_1{:}\kappa_1,\ldots,\beta_n{:}\kappa_n$ then $\Delta(\beta_i)$ is κ_i.

When I wish to refer to a component (either ϵ or $\{\eta \mapsto \tau\}$) of a store type generically, I will use the meta-variable a to range over components. Hence, in general, a store type C has the form $\emptyset * a_1 * \cdots * a_n$. I will continue to omit the initial "$\emptyset$" when a constraint is non-empty. For example, I write $\{\eta \mapsto \tau\}$ instead of $\emptyset * \{\eta \mapsto \tau\}$.

$$
\begin{array}{llll}
\textit{kinds} & \kappa & ::= & \texttt{Loc} \mid \texttt{Store} \mid \texttt{Type} \mid (\kappa_1, \ldots, \kappa_n) \to \texttt{Type} \\
\textit{constructor vars} & \beta & ::= & \zeta \mid \epsilon \mid \alpha \\
\textit{constructor ctxts} & \Delta & ::= & \cdot \mid \Delta, \beta{:}\kappa \\
\\
\textit{constructors} & c & ::= & r \mid C \mid \tau \\
\textit{locations} & \eta & ::= & \zeta \mid \ell \\
\textit{constraints} & C & ::= & \emptyset \mid C * \{r \mapsto \tau\} \mid C * \epsilon \\
\textit{types} & \tau & ::= & \alpha \mid \textit{junk} \mid \mathcal{S}(i) \mid \textit{ptr}(r) \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \tau_1 \cup \tau_2 \mid \\
& & & \forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0} \mid \exists[\Delta \mid C].\tau \mid \\
& & & \texttt{rec}\, \alpha\, (\Delta).\tau \mid c\, (c_1, \ldots, c_n)
\end{array}
$$

Figure 3.1: Alias Types: Syntax, Kinds & Constructors

As before, I use the notation $A[X/x]$ to denote the capture-avoiding substitution of X for a variable x in A. Occasionally, I use the notation $X[c_1, \ldots, c_n/\Delta]$ to denote capture-avoiding substitution of constructors $c_1, \ldots, c_n$ for the corresponding type variables in the domain of Δ.

Substitution is defined in the stardard way in all cases except for the substitution of constraints in constraints. To see why this case is different, consider substituting a constraint C' for ϵ in $C * \epsilon$. Assuming C does not contain ϵ, the natural result is $C * (C')$, but this store type is not syntactically well-formed: C' should be a single store component (say, a).[5] The result we want from this substitution is the list of the elements of C' appended to the list of elements C. I overload the $*$ symbol to denote the append operation. Thus,

$$(\emptyset * a_1 * \cdots * a_m) * (\emptyset * a'_1 * \cdots * a'_n) \stackrel{\text{def}}{=} \emptyset * a_1 * \cdots * a_m * a'_1 * \cdots * a'_n$$

Using the overloaded $*$ symbol, the definition of substitution of constraints in constraints is straightforward:

$$
\begin{array}{lcl}
\emptyset[C'/\epsilon] & \stackrel{\text{def}}{=} & \emptyset \\
(C * \{\eta \mapsto \tau\})[C'/\epsilon] & \stackrel{\text{def}}{=} & (C[C'/\epsilon]) * \{\eta \mapsto \tau[C'/\epsilon]\} \\
(C * \epsilon)[C'/\epsilon] & \stackrel{\text{def}}{=} & (C[C'/\epsilon]) * C'
\end{array}
$$

Type Well-Formedness Since types possess free variables of different kinds, not all syntactically well-formed type constructors make sense. Consequently, I introduce a new typing judgement to specify the meaningful type constructors. The judgement $\Delta \vdash_A c : \kappa$ states that a type constructor c is well-formed and has kind κ according to the assignment of free type variables to kinds given by Δ. The inference rules for this judgement are quite straightforward. Figures 3.2 and 3.3 contain the details.

Type Equality The type equality relation required for this type system is also somewhat more complex than those we have seen before. Types that differ only in the names of bound variables are equal to one another as before. In addition, store types are equal up to reordering

[5] I could change the syntax and allow $C * C'$ to be a syntactically well-formed store type. This decision makes the proofs slightly trickier. This is the approach I will take in the next chapter where capabilites have a more complex algebra.

$\boxed{\Delta \vdash_A c : \kappa}$

$$\frac{}{\Delta \vdash_A \ell : \texttt{Loc}} \ (\texttt{AT-}\ell)$$

$$\frac{}{\Delta \vdash_A \emptyset : \texttt{Store}} \ (\texttt{AT-empty}) \qquad \frac{\Delta \vdash_A C : \texttt{Store} \quad \Delta \vdash_A \epsilon : \texttt{Store}}{\Delta \vdash_A C * \epsilon : \texttt{Store}} \ (\texttt{AT-}\epsilon)$$

$$\frac{\Delta \vdash_A C : \texttt{Store} \quad \Delta \vdash_A \eta : \texttt{Loc} \quad \Delta \vdash_A \tau : \texttt{Type}}{\Delta \vdash_A C * \{\eta \mapsto \tau\} : \texttt{Store}} \ (\texttt{AT-pt})$$

Figure 3.2: Alias Types: Well-formedness, Locations & Store Types

of their components. A recursive type is not considered equal to its unfolding—objects of recursive type are explicitly unfolded using term-level coercions (see section 3.2.2).

The judgement $\Delta \vdash_A c_1 = c_2 : \kappa$ states that type constructors c_1 and c_2 are equivalent at kind κ. The type context Δ is included in the judgement so that well-formedness of the two constructors can be checked at the same time as equality. Formally:

Proposition 11 *If* $\Delta \vdash_A c_1 = c_2 : \kappa$ *then* $\Delta \vdash_A c_1 : \kappa$ *and* $\Delta \vdash_A c_2 : \kappa$.

The only interesting rule is the exchange rule (notice the rule checks well-formedness so the proposition holds):

$$\frac{\Delta \vdash_A C_1 * a_1 * a_2 * C_2 : \texttt{Store}}{\Delta \vdash_A C_1 * a_1 * a_2 * C_2 = C_1 * a_2 * a_1 * C_2 : \texttt{Store}} \ (\texttt{AT-exchange})$$

The equality relation also includes the obvious rules for reflexivity, symmetry, transitivity and congruence.

3.2.2 Small Values and Expressions

Figure 3.4 describes the syntax of the expression language. Expressions contain *small values*, the values that need not be allocated on the heap such as pointers and code[6], and a language of *coercions* that directs the type checker. Each of these components is described in the following paragraphs. A formal explanation of large, heap-allocated values such as memory blocks is deferred to section 3.2.3 where I also define programs and their operational semantics.

Small Values Small values are those values that are easily copied. They fit inside a single field of a memory block and may be passed directly to functions. In chapter two, pointers and junk were the only small values. In this chapter, I have extended this class considerably to allow us to write more realistic programs. There is junk (_) as before. We also have integers that are written ($\mathcal{S}(i)$) to indicate they will be given the specific singleton type $\mathcal{S}(i)$ rather than a standard (non-specific) integer type. Singleton integers will be used to construct the

[6]Recall we will not concern ourselves with garbage collecting code in this chapter and therefore will treat code as a small value.

$\boxed{\Delta \vdash_A c : \kappa}$

$$\frac{}{\Delta \vdash_A \beta : \Delta(\beta)}\ (\texttt{AT-}\beta) \qquad \frac{}{\Delta \vdash_A \mathit{junk} : \texttt{Type}}\ (\texttt{AT-junk})$$

$$\frac{}{\Delta \vdash_A \mathcal{S}(i) : \texttt{Type}}\ (\texttt{AT-sing}) \qquad \frac{\Delta \vdash_A \eta : \texttt{Loc}}{\Delta \vdash_A \mathit{ptr}(\eta) : \texttt{Type}}\ (\texttt{AT-ptr})$$

$$\frac{\Delta \vdash_A \tau_1 : \texttt{Type} \quad \cdots \quad \Delta \vdash_A \tau_n : \texttt{Type}}{\Delta \vdash_A \langle \tau_1, \ldots, \tau_n \rangle : \texttt{Type}}\ (\texttt{AT-tuple})$$

$$\frac{\Delta \vdash_A \tau_1 : \texttt{Type} \quad \Delta \vdash_A \tau_2 : \texttt{Type}}{\Delta \vdash_A \tau_1 \cup \tau_2 : \texttt{Type}}\ (\texttt{AT-union})$$

$$\frac{\begin{array}{c} \Delta, \Delta' \vdash_A C : \texttt{Store} \\ \Delta, \Delta' \vdash_A \tau_1 : \texttt{Type} \qquad \cdots \qquad \Delta, \Delta' \vdash_A \tau_n : \texttt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset) \end{array}}{\Delta \vdash_A \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0} : \texttt{Type}}\ (\texttt{AT-fun})$$

$$\frac{\begin{array}{c} \Delta, \Delta' \vdash_A C : \texttt{Store} \qquad \Delta, \Delta' \vdash_A \tau : \texttt{Type} \\ (\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset) \end{array}}{\Delta \vdash_A \exists[\Delta' \mid C].\tau : \texttt{Type}}\ (\texttt{AT-}\exists)$$

$$\frac{\begin{array}{c} \Delta, \alpha{:}(\kappa_1, \ldots, \kappa_n) \to \texttt{Type}, \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \vdash_A \tau : \texttt{Type} \\ (\text{for } 1 \leq i \leq n. \beta_i \notin \mathit{Dom}(\Delta)) \end{array}}{\Delta \vdash_A \texttt{rec}\, \alpha\, (\beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n).\tau : (\kappa_1, \ldots, \kappa_n) \to \texttt{Type}}\ (\texttt{AT-rec})$$

$$\frac{\Delta \vdash_A c : (\kappa_1, \ldots, \kappa_n) \to \texttt{Type} \quad \Delta \vdash_A c_1 : \kappa_1 \quad \cdots \quad \Delta \vdash_A c_n : \kappa_n}{\Delta \vdash_A c\,(c_1, \ldots, c_n) : \texttt{Type}}\ (\texttt{AT-rec-app})$$

Figure 3.3: Alias Types: Well-formedness, Types

$$
\begin{array}{llll}
\textit{small values} & v & ::= & x \mid _ \mid \mathcal{S}(i) \mid \mathtt{ptr}(\ell) \mid \mathtt{fix}\ f\ [\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid v[c] \\
\textit{coercions} & \gamma & ::= & \mathtt{union}_{\tau_1 \cup \tau_2}(r) \mid \mathtt{roll}_{\mathtt{rec}\,\alpha\,(\Delta).\tau\,(c_1,\ldots,c_n)}(r) \mid \mathtt{unroll}(r) \mid \\
 & & & \mathtt{pack}_{[c_1,\ldots,c_n|C]\,\mathtt{as}\,\exists[\Delta|C].\tau}(r) \mid \mathtt{unpack}\,r\,\mathtt{with}\,\Delta \\
\textit{expressions} & e & ::= & \mathtt{let}\,\rho, x = \mathtt{new}\ (i)\ \mathtt{in}\ e \mid \mathtt{free}\ v; e \mid \\
 & & & \mathtt{let}\ x = v.i\ \mathtt{in}\,e \mid v_1.i{:=}v_2; e \mid \mathtt{if}\ v\ (e_1 \mid e_2) \mid \\
 & & & v(v_1, \ldots, v_n) \mid \mathtt{halt}\ v \mid \\
 & & & \mathtt{coerce}(\gamma); e
\end{array}
$$

Figure 3.4: Alias Types: Syntax, Expressions

data types described in the previous section. The second kind of singleton value is the pointer to location ℓ, written $\mathtt{ptr}(\ell)$.

Functions are much more exciting than in the last chapter as they may be both polymorphic and recursive. Polymorphism is necessary to promote code reuse and recursion allows us to construct and traverse recursive data types. Recursive functions have the form

$$\mathtt{fix}\ f\ [\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$$

The type context Δ binds free variables in the store type C, argument types τ_1 through τ_n, and body of the function. The function has type

$$\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}$$

and f (the recursive function name) is given this type in the function body. In other words, these functions support polymorphic recursion, a feature that is critical for coding many practical examples (See section 3.4).

In order to keep type checking simple, I use explicit syntax for instantiating polymorphic types. If v is a polymorphic function with type

$$\forall[\beta{:}\kappa, \Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}$$

then the explicit instantiation $v[c]$[7] has type

$$\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}[c/\beta]$$

Notice that partial type instantiation is allowed.

Figure 3.5 presents the static semantics of small values. The judgements have the form $\Delta; \Gamma \vdash_A v : \tau$ where Δ contains the free type variables of types in Γ and v. The value context Γ is handled somewhat differently here than before. In the previous chapter, I took special care to lay out all of the structural rules for contexts (exchange for linear values, weakening and contraction for intuitionistic ones). However, here, I have engineered the expression language so that only small values are bound to variables. This is why variables themselves are considered small. Since small types are cheap, I treat them intuitionistically and consequently, the entire

[7]The form $v[c]$ is still considered a value because type constructors have no real run-time representation—they may be erased before executing a program. Thus at run time, after types have been erased, $v[c]$ is represented as v. In section 3.2.3, I will give a typed operational semantics, but it can easily be proven equivalent to a semantics in which types have been erased. The typed operational semantics simplifies the proof of type soundness.

$\boxed{\Delta; \Gamma \vdash_A v : \tau}$

$$\frac{}{\Delta; \Gamma \vdash_A x : \Gamma(x)} \; (\texttt{A-var})$$

$$\frac{}{\Delta; \Gamma \vdash_A _ : \mathit{junk}} \; (\texttt{A-junk})$$

$$\frac{}{\Delta; \Gamma \vdash_A \mathcal{S}(i) : \mathcal{S}(i)} \; (\texttt{A-i})$$

$$\frac{}{\Delta; \Gamma \vdash_A \mathtt{ptr}(\ell) : \mathit{ptr}(\ell)} \; (\texttt{A-ptr})$$

$$\frac{\begin{array}{c}\Delta \vdash_A \forall[\Delta'].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0} = \tau_f : \mathtt{Type} \\ \Delta\Delta'; \Gamma, f{:}\tau_f, x_1{:}\tau_1, \ldots, x_n{:}\tau_n; C' \vdash_A e\end{array}}{\Delta; \Gamma \vdash_A \mathtt{fix}\ f\ [\Delta'](C', x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \tau_f} \; (\texttt{A-fix})$$

$$\frac{\Delta; \Gamma \vdash_A v : \forall[\beta{:}\kappa, \Delta'].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0} \quad \Delta \vdash_A c : \kappa}{\Delta; \Gamma \vdash_A v[c] : (\forall[\Delta'].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0})[c/\beta]} \; (\texttt{A-tapp})$$

$$\frac{\Delta; \Gamma \vdash_A v : \tau' \quad \Delta \vdash_A \tau' = \tau : \mathtt{Type}}{\Delta; \Gamma \vdash_A v : \tau} \; (\texttt{A-veq})$$

Figure 3.5: Alias Types: Static Semantics, Values

value context Γ is also intuitionistic, just the opposite of the pure linear languages. Hence, rather than spell out all of the structural rules for a purely intuitionistic context, I prefer to use a more standard treatment and assume that Γ is simply a finite partial map from variables to types. In the typing rule for variables, there is no restriction that the context contain only a single binding. Similarly, in the rules for junk and singletons, there may be unlimited amounts of left-over bindings in the context—the system does not attempt to garbage collect small values. Aside from these points, the rules are as one might expect.

Expressions Figure 3.6 present the typing rules for expressions. The judgement $\Delta; \Gamma; C \vdash_A e$ states that in type context Δ, a store described by C and value context Γ, the expression e is well-formed. Most of the expressions have similar operational interpretations in this language as in the linear languages of the previous chapter, but the typing rules are considerably more complex due to the necessity of tracking dependencies between memory and pointers.

Memory Management Expressions Consider the `new` expression. It allocates a memory block containing i junk fields at a fresh location ℓ. A pointer to the location is substituted for x and ℓ is substituted for the location variable ζ. This operation is modeled in the type system (see rule `A-new`) by extending the store type with a memory block type of length i.

$\boxed{\Delta; \Gamma; C \vdash_A e}$

$$\frac{\Delta, \rho{:}\texttt{Loc}; \Gamma, x{:}ptr(\rho); C * \{\rho \mapsto \overbrace{\langle junk, \ldots, junk \rangle}^{i}\} \vdash_A e}{\Delta; \Gamma; C \vdash_A \texttt{let}\, \rho, x = \texttt{new}\ (i)\ \texttt{in}\ e} \begin{pmatrix} x \notin Dom(\Gamma) \\ \rho \notin Dom(\Delta) \end{pmatrix} \ (\texttt{A-new})$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash_A v : ptr(\zeta) \\ \Delta \vdash_A C = C' * \{\zeta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} : \texttt{Store} \qquad \Delta \vdash_A C' \Gamma e \end{array}}{\Delta; \Gamma; C \vdash_A \texttt{free}\ v; e} \ (\texttt{A-free})$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash_A v : ptr(\zeta) \\ \Delta \vdash_A C = C' * \{\zeta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} : \texttt{Store} \\ \Delta; \Gamma, x{:}\tau_i; C \vdash_A e \end{array}}{\Delta; \Gamma; C \vdash_A \texttt{let}\ x = v.i\ \texttt{in}\, e} \begin{pmatrix} 1 \leq i \leq n \\ x \notin Dom(\Gamma) \end{pmatrix} \ (\texttt{A-pi})$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash_A v_1 : ptr(\zeta) \qquad \Delta; \Gamma \vdash_A v_2 : \tau \\ \Delta \vdash_A C = C' * \{\zeta \mapsto \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle\} : \texttt{Store} \\ \Delta; \Gamma; C' * \{\zeta \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\} \vdash_A e \end{array}}{\Delta; \Gamma; C \vdash_A v_1.i{:=}v_2; e} \ (1 \leq i \leq n) \ (\texttt{A-ai})$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash_A v : ptr(\zeta) \qquad \Delta \vdash_A C = C' * \{\zeta \mapsto \tau_1 \cup \tau_2\} : \texttt{Store} \\ \Delta \vdash_A \tau_1 = \exists[\Delta'_1 \mid C'_1]. \cdots \exists[\Delta'_j \mid C'_j].\langle \mathcal{S}(1), \tau'_1, \ldots, \tau'_k \rangle : \texttt{Type} \\ \Delta \vdash_A \tau_2 = \exists[\Delta''_1 \mid C''_1]. \cdots \exists[\Delta''_m \mid C''_m].\langle \mathcal{S}(2), \tau''_1, \ldots, \tau''_n \rangle : \texttt{Type} \\ \Delta; \Gamma; C' * \{\zeta \mapsto \tau_1\} \vdash_A e_1 \qquad \Delta; \Gamma; C' * \{\zeta \mapsto \tau_2\} \vdash_A e_2 \end{array}}{\Delta; \Gamma; C \vdash_A \texttt{if}\ v\ (e_1 \mid e_2)} \ (\texttt{A-case})$$

$$\frac{\Delta; \Gamma \vdash_A v : (C, \tau_1, \ldots, \tau_n) \to \mathbf{0} \quad \Delta; \Gamma \vdash_A v_1 : \tau_1 \quad \cdots \quad \Delta; \Gamma \vdash_A v_n : \tau_n}{\Delta; \Gamma; C \vdash_A v(v_1, \ldots, v_n)} \ (\texttt{A-app})$$

$$\frac{\Delta; \Gamma \vdash_A v : ptr(\eta) \quad \Delta \vdash_A C = \{\eta \mapsto \langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle\} : \texttt{Store}}{\Delta; \Gamma; C \vdash_A \texttt{halt}\ v} \ (\texttt{A-halt})$$

Figure 3.6: Alias Types: Static Semantics, Instructions

Once a block has been allocated, it may be operated on by accessor functions $\texttt{let}\ x = v_1.i\ \texttt{in}\ e$ and $v_1.i{:=}v_2; e$, which project from or store into the ith field of v_1. The projection operation (typed by rule `A-pi`) is well-formed if v_1 is a pointer to some location η and that location contains a object with type $\langle\tau_1, \ldots, \tau_n\rangle$ (where i is less than n). In this case, the following expression e must be well-formed given the additional assumption that x has type τ_i. The update operation (typed by rule `A-ai`) is similar to the projection operation in that v_1 must be a pointer to a location containing a memory block. However, e is verified in a context where the type of the memory block has changed: The ith field has type τ where τ is the type of the object being stored into that location, but is otherwise unconstrained.

As a concrete example, consider the process of allocating and initializing a pair of pairs, where the deeper pair is aliased. The comments on the right-hand side present a portion of the type checking context after each program point.

$\texttt{let}\ \zeta_x, x = \texttt{new}\ (2)\ \texttt{in}$	% $x{:}ptr(\zeta_x)$ % $\{\zeta_x \mapsto \langle junk, junk\rangle\}$
$\texttt{let}\ \zeta_y, y = \texttt{new}\ (2)\ \texttt{in}$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle junk, junk\rangle\} * \{\zeta_y \mapsto \langle junk, junk\rangle\}$
$x.1{:=}y;$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle ptr(\zeta_y), junk\rangle\} * \{\zeta_y \mapsto \langle junk, junk\rangle\}$
$x.2{:=}y;$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle ptr(\zeta_y), ptr(\zeta_y)\rangle\} * \{\zeta_y \mapsto \langle junk, junk\rangle\}$
$\vdots$	

At each update operation, the type checker verifies that x has a pointer type and modifies the type of x's memory block accordingly. The interesting aspect of this example is that after the fourth instruction in the sequence, there are three aliases to the second memory block: the variable y and the two components of x. We can see this is true, simply by counting the number of occurences of $ptr(\zeta_y)$ in the type checking context. Each occurence must alias the others. All three aliases are accurately tracked in the type system and any of them may be used. The code could even create a fourth alias z by projecting one of the components x and use it to initialize the second pair:

$\texttt{let}\ z = x.1\ \texttt{in}$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y), z{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle ptr(\zeta_y), ptr(\zeta_y)\rangle\} * \{\zeta_y \mapsto \langle junk, junk\rangle\}$
$z.1{:=}3;$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y), z{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle ptr(\zeta_y), ptr(\zeta_y)\rangle\} * \{\zeta_y \mapsto \langle int, junk\rangle\}$
$z.2{:=}3;$	% $x{:}ptr(\zeta_x), y{:}ptr(\zeta_y), z{:}ptr(\zeta_y)$ % $\{\zeta_x \mapsto \langle ptr(\zeta_y), ptr(\zeta_y)\rangle\} * \{\zeta_y \mapsto \langle int, int\rangle\}$
$\vdots$	

To understand formally where we obtain extra flexibility over a linear type system, recall the linear CPS rule for projection:

$$\frac{\Gamma, x{:}\tau_1, y{:}\mathit{junk} \otimes \tau_2 \vdash_{cps} e}{\Gamma, y{:}\tau_1 \otimes \tau_2 \vdash_{cps} \mathtt{let}\, x = y.1 \,\mathtt{in}\, e} \begin{pmatrix} x \notin \mathit{Dom}(\Gamma) \cup \{y\} \\ \tau_1 \neq \mathit{junk} \end{pmatrix} \ (\mathtt{cps\text{-}p1})$$

The principle difference is that after projection, the first component in the linear rule becomes junk. This is not the case in rule `A-pi`: Both projected object x and component $v_1.i$ are useful aliases of one another. The reason this rule is admissible here, but not in the linear type system is that our types are much more precise. Singleton types allow the type checker to keep track of all aliases.

A second subtle difference is that the rule `A-pi` no longer prevents projection of junk components; there is no side condition asserting that τ_i is not equal to *junk*. In the presence of polymorphism, we cannot give this guarantee—at least, not without other complications. Still, as stated in chapter two, this side condition is not essential for soundness. The important constraint for soundness is that junk is not dereferenced or used as a function and the other typing rules ensure this does not happen. However, because the typing rule is a little more lax here than in the linear type system, in some cases, an error may be reported later in the program text (errors are flagged when junk is used in the alias type system) than it would be otherwise (errors are flagged when junk is first projected, which may be well before it is used in the linear type system).

Rule `A-ai` also bears an interesting relation to the analogous rule for the linear CPS language (rule `cps-a1`):

$$\frac{\Gamma, x{:}\tau_1 \otimes \tau_2 \vdash_{cps} e}{\Gamma, x{:}\mathit{junk} \otimes \tau_2, y{:}\tau_1 \vdash_{cps} x.1{:=}y; e} \ (\mathtt{cps\text{-}a1})$$

When verifying the following expression e, the rule `cps-a1` deletes the binding for y from the context Γ so y can no longer be used. In contrast, the rule `A-ai` allows the value that is assigned to be used repeatedly. If that value is a pointer, then the assignment copies it, creating an alias. Fortunately, the dependencies in the type system track these aliases.

Rule `cps-a1` also requires that the first component of the pair being stored into is junk. This constraint ensures that no object is made unreachable due to component assignment and it made it possible to prove the Complete Collection theorem for the linear type system. Surprisingly, the new type system can be less restrictive than the linear type system. Even if the first component of a pair holds a valid pointer, it need not be the only pointer to a particular location—it may have many aliases—so the rule `A-ai` allows the assignment. Of course, we now run into a problem. What if the assignment does, in fact, overwrite the last pointer to some heap-allocated data structure? There is nothing in the rule `A-ai` that prevents this possibility and it would cause the data structure to become unreachable and uncollectable. Does the alias type system provide any guarantees about the amount of garbage that a program collects?

In fact, the type system does provide formal guarantees about how much garbage is collected, although the guarantees have a slightly different quality than those for the linear type system and the mechanism for enforcing them is different as well. The key is in the rule `A-halt` for the terminal expression. It requires programs terminate with a store that can be typed by the constraints

$$C_{halt} = \{\eta \mapsto \langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle\}$$

The only stores that satisfy this constraint are ones with exactly one location (η). Moreover, that one location must contain a boolean. In earlier sections, I looked upon stores of this form

unfavourably — they are not very polymorphic. Without any store polymorphism, these types characterize very few stores. However, that is exactly what we need here. It means that if a program terminates, the store cannot contain any garbage. For example, suppose a programmer forgets to deallocate a reference cell, but otherwise writes a type-safe program. At the program point immediately preceding the halt instruction, the store will have the type

$$\{\eta \mapsto \langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle\} * \{\eta' \mapsto \langle \tau \rangle\}$$

When checking the halt instruction, the type checker will flag an error since the weakening rule is not valid for store types:

$$\{\eta \mapsto \langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle\} * \{\eta' \mapsto \langle \tau \rangle\} \neq C_{halt}$$

This design decision has several ramifications for the garbage collection properties of the alias type system. As was the case for the projection rule, the additional flexibility in the rule for component update may cause the type checker to flag certain program points as erroneous (the halt expression) when the actual logical error occurred much earlier (the last pointer to an object is over-written without freeing the object). However, the situation is perhaps more grave here as non-terminating programs have no constraint on how much unreachable storage they may create. As an example, consider the parameterless function `loop`:

$$\texttt{fix loop}\ [\epsilon](\epsilon,).\texttt{let}\ \zeta, x = \texttt{new}\ (1)\ \texttt{in}\ \texttt{loop}[\epsilon * \{\zeta \mapsto \mathit{junk}\}]()$$

The loop may be invoked in any state, simply by instantiating the polymorphic variable ϵ with the current store type. Each time around the loop, another useless reference cell is allocated and then the only pointer to it is discarded.

The last memory management operation is `free` $v; e$. It deallocates the memory block pointed to by v. This effect is reflected in the typing rule for free by requiring that the following expression be well-formed in a context C' that does not include the location in question. Again, the presence of aliasing makes the `free` expression more interesting in this language than in the linear language. The pointer being freed may have many aliases that suddenly become potentially dangerous dangling pointers. Fortunately, none of these dangling pointers will be useable; they may be copied and passed to functions, but never dereferenced. The rules for projection and assignment always verify that when dereferencing a pointer with type $\texttt{ptr}(\eta)$, the store is described by a type containing the mapping $\{\eta \mapsto \langle \tau_1, \ldots, \tau_n \rangle\}$, so that pointer cannot dangle.

As in the assignment rule, there are no restrictions on the types of components of the memory block that is freed. Since the free is shallow and the components of the object freed may contain pointers, deallocation can make an object unreachable. Once again, the formulation of the `A-halt` rule ensures that programmers clean up after themselves.

Other Expressions The `if` v $(e_1 \mid e_2)$ expression checks the first field of the memory block in the location pointed to by a value v. If the first field is a 1, execution continues with the first branch, and if it is a 2, execution continues with the second branch. The typing rule for the construct is somewhat complex as the memory type constructor $\langle \cdots \rangle$ will not be the top-most type constructor (otherwise, the case would be unnecessary). The type system expects a union type to be the top-most and each alternative may contain some number (possibly zero) of existential quantifiers to abstract the store encapsulated in that alternative. The underlying memory value must have either tag 1 or tag 2 in its first field. As mentioned earlier, it is

$\boxed{\Delta; \Gamma; C \vdash_A e}$

$$\frac{\Delta; C \vdash_A \gamma \Longrightarrow \Delta'; C' \quad \Delta'; C'; \Gamma \vdash_A e}{\Delta; \Gamma; C \vdash_A \mathtt{coerce}(\gamma); e} \ (\mathtt{A\text{-}coerce})$$

$\boxed{\Delta; C \vdash_A \gamma \Longrightarrow \Delta'; C'}$

$$\frac{\Delta \vdash_A \tau_1 : \mathtt{Type} \qquad \Delta \vdash_A \tau_2 : \mathtt{Type} \qquad \Delta \vdash_A C = C' * \{r \mapsto \tau_i\} : \mathtt{Store}}{\Delta; C \vdash_A \mathtt{union}_{\tau_1 \cup \tau_2}(r) \Longrightarrow \Delta; C' * \{r \mapsto \tau_1 \cup \tau_2\}} \ (\text{for } i = 1 \text{ or } 2) \ (\mathtt{A\text{-}}\cup)$$

$$\frac{\Delta \vdash_A \tau = (\mathtt{rec}\,\alpha\,(\Delta').\tau')\,(c_1, \ldots, c_n) : \mathtt{Type} \qquad \Delta \vdash_A C = C' * \{r \mapsto \tau'[\mathtt{rec}\,\alpha\,(\Delta').\tau'/\alpha][c_1, \ldots, c_n/\Delta']\} : \mathtt{Store}}{\Delta; C \vdash_A \mathtt{roll}_\tau(r) \Longrightarrow \Delta; C' * \{r \mapsto \tau\}} \ (\mathtt{A\text{-}roll})$$

$$\frac{\Delta \vdash_A C = C' * \{r \mapsto \tau\} : \mathtt{Store} \qquad \Delta \vdash_A \tau = (\mathtt{rec}\,\alpha\,(\Delta').\tau')\,(c_1, \ldots, c_n) : \mathtt{Type}}{\Delta; C \vdash_A \mathtt{unroll}(r) \Longrightarrow \Delta; C' * \{r \mapsto \tau'[\mathtt{rec}\,\alpha\,(\Delta').\tau'/\alpha][c_1, \ldots, c_n/\Delta']\}} \ (\mathtt{A\text{-}unroll})$$

$$\frac{\Delta' = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \qquad \cdot \vdash_A c_i : \kappa_i \quad (\text{for } 1 \le i \le n) \qquad \Delta \vdash_A C = C'' * \{r \mapsto \tau[c_1, \ldots, c_n/\Delta']\} * C'[c_1, \ldots, c_n/\Delta'] : \mathtt{Store}}{\Delta; C \vdash_A \mathtt{pack}_{[c_1, \ldots, c_n | C'[c_1, \ldots, c_n/\Delta']]\,\mathtt{as}\,\exists[\Delta' | C'].\tau}(r) \Longrightarrow \Delta; C'' * \{r \mapsto \exists[\Delta' \mid C'].\tau\}} \ (\mathtt{A\text{-}pack})$$

$$\frac{\Delta \vdash_A C = C'' * \{r \mapsto \exists[\Delta' \mid C'].\tau\} : \mathtt{Store}}{\Delta; C \vdash_A \mathtt{unpack}\,r\,\mathtt{with}\,\Delta' \Longrightarrow \Delta, \Delta'; C'' * \{r \mapsto \tau\} * C'} \ (\mathtt{A\text{-}unpack})$$

Figure 3.7: Alias Types: Static Semantics, Coercions

$$
\begin{array}{llll}
\textit{storable val's} & h & ::= & \langle v_1, \ldots, v_n \rangle \mid \varsigma(h) \\
\textit{witnesses} & \varsigma & ::= & \texttt{union}_{\tau_1 \cup \tau_2} \mid \texttt{pack}_{[c_1,\ldots,c_n|S]\texttt{as}\exists[\Delta|C].\tau} \mid \texttt{roll}_{(\texttt{rec}\,\alpha\,(\Delta).\tau)\,(c_1,\ldots,c_n)} \\
\textit{stores} & S & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\textit{programs} & P & ::= & (S, e)
\end{array}
$$

Figure 3.8: Alias Types: Syntax, Stores & Programs

possible to formulate other union elimination constructs including pointer equality checks or discrimination between pointers and small integers (such as null implemented by 0).

Function calls are handled in the standard way and justification for the `halt` expression has been given above.

The last expression $\texttt{coerce}(\gamma)$ applies a typing coercion to the store. Coercions, unlike the other expressions are for type-checking purposes only. Intuitively, coercions may be erased before executing a program and the run-time behaviour will not be effected. The judgement form $\Delta; C \vdash_A \gamma \Longrightarrow \Delta'; C'$ indicates that a coercion is well-formed, extends the type context to Δ', and produces new store constraints C'. These judgements are presented in figure 3.7.

Each coercion operates on a particular store location η. The `union` coercion lifts the object at η into a union type and the `roll`/`unroll` coercions witness the isomorphism between a recursive type and its unfolding. The coercion

$$\texttt{pack}_{[c_1,\ldots,c_n|C'[c_1,\ldots,c_n/\Delta']]\texttt{as}\,\exists[\Delta'|C'].\tau}(\eta)$$

introduces an existential type by hiding the type constructors $c_1, \ldots, c_n$ and encapsulating the store described by $C'[c_1, \ldots, c_n/\Delta']$. It is critical that the constraints $C'[c_1, \ldots, c_n/\Delta']$ be removed from the context when verifying the instructions that follow the pack coercion. Otherwise there would be two copies of the constraints (one in the top-level type-checking context and one in the existential) and the linear nature of the store typing would be broken. The `unpack` coercion eliminates an existential type, augments the current constraints with the encapsulated C', and extends the type context Δ with Δ', the hidden type constructors.

3.2.3 Stores and Programs

As before, programs are pairs of a store and an expression. Stores are finite partial maps from concrete locations to storable values. Figure 3.8 defines the syntax of stores and programs.

Storable Values The storable values consist of memory blocks $\langle v_1, \ldots, v_n \rangle$ and witnessed values $\varsigma(h)$. Notice that the components of memory blocks must be small values. As mentioned in the previous section, projection operations copy the components of memory blocks. In order to ensure that this copying is cheap and requires no allocation, memory block components must be small. Witnessed values are introduced by coercions. For example, the union coercion introduces a union witness and similarly for roll and pack coercions. These witnesses have no effect on the run-time behaviour of well-typed programs; their purpose is to keep type checking (almost) syntax-directed.

Figure 3.9 presents the static semantics of storable values. These values only appear during evaluation of a program and therefore they are always closed. As result, the typing judgements require no context to verify that storable values are well-formed. Aside from this fact, the rules

$$\frac{\cdot;\cdot \vdash_A v_1 : \tau_1 \quad \cdots \quad \cdot;\cdot \vdash_A v_n : \tau_n}{\vdash_A \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \text{ (AS-block)}$$

$$\frac{\cdot \vdash_A \tau_1 \cup \tau_2 : \texttt{Type} \qquad \vdash_A h : \tau_1 \quad \text{or} \quad \vdash_A h : \tau_2}{\vdash_A \texttt{union}_{\tau_1 \cup \tau_2}(h) : \tau_1 \cup \tau_2} \text{ (AS-}\cup\text{)}$$

$$\frac{\begin{array}{c}\cdot \vdash_A \tau = (\texttt{rec}\,\alpha\,(\Delta).\tau')\,(c_1, \ldots, c_n) : \texttt{Type} \\ \vdash_A h : \tau'[\texttt{rec}\,\alpha\,(\Delta).\tau'/\alpha][c_1, \ldots, c_n/\Delta]\end{array}}{\vdash_A \texttt{roll}_\tau(h) : \tau} \text{ (AS-rec)}$$

$$\frac{\begin{array}{c}\Delta = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \qquad \cdot \vdash_A c_i : \kappa_i \quad (\text{for } 1 \leq i \leq n) \\ \vdash_A S : C[c_1, \ldots, c_n/\Delta] \qquad \vdash_A h : \tau[c_1, \ldots, c_n/\Delta]\end{array}}{\vdash_A \texttt{pack}_{[c_1,\ldots,c_n|S]\texttt{as}\exists[\Delta|C].\tau}(h) : \exists[\Delta \mid C].\tau} \text{ (AS-}\exists\text{)}$$

$$\frac{\vdash_A h : \tau' \quad \cdot \vdash_A \tau' = \tau : \texttt{Type}}{\vdash_A h : \tau} \text{ (AS-eq)}$$

Figure 3.9: Alias Types: Static Semantics, Storable Values

for storable values are mostly unremarkable. Memory blocks have type $\langle \tau_1, \ldots, \tau_n \rangle$ and the witnesses coerce values of a particular type into unions, existentials or recursive types as one might expect.

Stores The store well-formedness judgement is written $\vdash_A S : C$ and is given below.

$$\frac{\begin{array}{c}S = \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\ \cdot \vdash_A C = \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\} : \texttt{Store} \\ \vdash_A h_1 : \tau_1 \qquad \cdots \qquad \vdash_A h_n : \tau_n\end{array}}{\vdash_A S : C} \text{ (A-store)}$$

Unlike in the typing rules for linear stores, the locations in the domain of the store can appear many times in the storable values h, making it possible to represent shared data structures and cycles.

There is one important restriction, called *Global Uniqueness*, on where locations may appear that is not captured in the store typing rule. It states that there can be no duplication of locations in the domain of the store or in any encapsulated store:

Definition 12 (Global Uniqueness) $\texttt{GU}(S)$ *if and only if there are no duplicate locations in* $\texttt{L}(S)$.

Definition 13 (Global Store Locations) $\texttt{L}(S)$ *is the multi-set given by the following defi-*

nition.

$$
\begin{array}{lcl}
\mathtt{L}(\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}) & = & \{\ell_1, \ldots, \ell_n\} \uplus \mathtt{L}(h_1) \uplus \cdots \uplus \mathtt{L}(h_n) \\
\mathtt{L}(\mathtt{pack}_{[c_1,\ldots,c_n|S]\mathtt{as}\tau}(h)) & = & \mathtt{L}(S) \uplus \mathtt{L}(h) \\
\\
\mathtt{L}(x) & = & \mathtt{L}(x_1) \uplus \cdots \uplus \mathtt{L}(x_n)
\end{array}
$$

for any other term construct x
where $x_1, \ldots, x_n$ are the subcomponents of x.

Programs A program is well-formed, written $\vdash_A (S, e)$, under the following circumstances.

Definition 14 (Well-formed Program) $\vdash_A (S, e)$ *iff*

1. *The store adheres to global uniqueness* $\mathtt{GU}(S)$.
2. *There exists constraints C such that $\vdash_A S : C$.*
3. *The expression is well-formed with the given constraints: $\cdot; \cdot; C \vdash_A e$.*

3.2.4 Operational Semantics

The small-step operational semantics for the language is given by a function $P \longmapsto_A P'$. The majority of the operational rules are entirely standard and formalize the intuitive rules described earlier in the chapter. The operational rule for the coercion expression depends upon a separate semantics for coercions that has the form $S \longmapsto_A S', \theta$ where θ is a substitution of type constructors for type constructor variables. Inspection of these rules reveals that coercions do not alter the association between locations and memory blocks; they simply insert witnesses that alter the typing derivation so that it is possible to prove a type soundness result. Figures 3.10 and 3.11. contain the rules for program and coercion operational semantics.

3.3 Properties of the Aliasing Language

As in the linear language, the two important properties of the alias type system are Type Soundness and Complete Collection.

3.3.1 Type Soundness

As before, I define the stuck program states and then use Subject Reduction and Progress lemmas to prove type soundness. In this language, a *stuck program* is a program that is not in the terminal configuration

$$(\{\ell \mapsto v\}, \mathtt{halt}\ \mathtt{ptr}(\ell))$$

where v is a boolean ($i.e.\ \mathtt{union}_{\langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2)\rangle}(\langle\mathcal{S}(i)\rangle)$) and for which no operational rule applies. The definitions of Subject Reduction and Progress lemmas and their proofs can be found in Appendix A.

Theorem 15 (Alias Type Soundness)
If $\vdash_A (S, e)$ and $(S, e) \longmapsto_A^* (S', e')$ then (S', e') is not stuck.

$$\boxed{(S,e) \longmapsto_A (S,e)}$$

$$(S, \texttt{let}\,\zeta, x = \texttt{new}\ (i)\ \texttt{in}\ e) \quad \longmapsto_A \quad (S\{\ell \mapsto h\}, e[\ell/\zeta][\texttt{ptr}(\ell)/x])$$
$$\text{where } \ell \notin S, e \text{ and } h = \overbrace{\langle _, \ldots, _ \rangle}^{i}$$

$$(S\{\ell \mapsto h\}, \texttt{free}\ \texttt{ptr}(\ell); e) \quad \longmapsto_A \quad (S, e)$$
$$\text{where} \quad h = \langle v_1, \ldots, v_n \rangle$$

$$(S\{\ell \mapsto h\}, \texttt{let}\ x = \texttt{ptr}(\ell).i\ \texttt{in}\ e) \quad \longmapsto_A \quad (S\{\ell \mapsto h\}, e[v_i/x])$$
$$\begin{aligned} \text{where} \quad & 1 \le i \le n \\ & h = \langle v_1, \ldots, v_i, \ldots, v_n \rangle \end{aligned}$$

$$(S\{\ell \mapsto h\}, \texttt{ptr}(\ell).i{:=}v'; e) \quad \longmapsto_A \quad (S\{\ell \mapsto h'\}, e)$$
$$\begin{aligned} \text{where} \quad & 1 \le i \le n \\ & h = \langle v_1, \ldots, v_i, \ldots, v_n \rangle \\ & h' = \langle v_1, \ldots, v', \ldots, v_n \rangle \end{aligned}$$

$$(S\{\ell \mapsto h\}, \texttt{if}\ \texttt{ptr}(\ell)\ (e_1 \mid e_2)) \quad \longmapsto_A \quad (S\{\ell \mapsto h'\}, e_i)$$
$$\begin{aligned} \text{where} \quad & i = 1 \text{ or } 2 \\ & h = \texttt{union}_{\tau_1 \cup \tau_2}(h') \\ & h' = \varsigma_1(\cdots \varsigma_m(\langle \mathcal{S}(i), v_1, \ldots, v_n \rangle) \cdots) \end{aligned}$$

$$(S, v(v_1, \ldots, v_n)) \quad \longmapsto_A \quad (S, \theta(e))$$
$$\begin{aligned} \text{where} \quad & v = v'[c_1, \ldots, c_m] \\ & v' = \texttt{fix}\ f\ [\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ & \theta = [c_1, \ldots, c_m/\Delta][v'/f][v_1, \ldots, v_n/x_1, \ldots, x_n] \end{aligned}$$

$$(S, \texttt{coerce}(\gamma); e) \quad \longmapsto_A \quad (S', \theta(e))$$
$$\text{where } \gamma(S) \longmapsto_A S', \theta$$

Figure 3.10: Alias Types: Operational Semantics, Programs

$\boxed{\gamma(S) \longmapsto_A S', \theta}$

$$\texttt{union}_{\tau_1 \cup \tau_2}(\ell)(S\{\ell \mapsto v\}) \quad \longmapsto_A \quad S\{\ell \mapsto \texttt{union}_{\tau_1 \cup \tau_2}(v)\}, []$$

$$\texttt{roll}_\tau(\ell)(S\{\ell \mapsto v\}) \quad \longmapsto_A \quad S\{\ell \mapsto \texttt{roll}_\tau(v)\}, []$$

$$\texttt{unroll}(\ell)(S\{\ell \mapsto \texttt{roll}_\tau(v)\}) \quad \longmapsto_A \quad S\{\ell \mapsto v\}, []$$

$$\texttt{pack}_{[c_1,\ldots,c_n|C]\texttt{as}\,\tau}(\ell)(S\{\ell \mapsto v\}S') \quad \longmapsto_A \quad S\{\ell \mapsto \texttt{pack}_{[c_1,\ldots,c_n|S']\texttt{as}\,\tau}(v)\}, []$$
where $C = \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m\}$ and $S' = \{\ell_1 \mapsto v_1, \ldots, \ell_m \mapsto v_m\}$

$$\texttt{unpack}\,\ell\,\texttt{with}\,\Delta\,(S\{\ell \mapsto \texttt{pack}_{[c_1,\ldots,c_n|S']\texttt{as}\,\exists[\Delta|C].\tau}(v)\}) \quad \longmapsto_A \quad SS'\{\ell \mapsto v\}, [c_1, \ldots, c_n/\Delta]$$

Figure 3.11: Alias Types: Operational Semantics, Coercions

3.3.2 Complete Collection

Complete Collection is formulated quite differently here than in the linear type system. Reachability characterizes the garbage collected by the linear type system accurately, but it is incomparable to the garbage collected by the alias type system. As I have demonstrated, non-terminating alias type programs can make memory unreachable. They can also collect data that is still reachable, leaving safe, but dangling pointers in data structures.

Instead of specifying garbage through reachability, the theorem specifies the termination behaviour of well-formed programs. Informally, if a well-formed program runs to completion, then it collects all stored data structures, save the single boolean result. This proof also follows from Subject Reduction and Progress lemmas. It can also be found in Appendix A.

Theorem 16 (Alias Type Complete Collection)
If $\vdash_A (S, e)$ *and* $(S, e) \longmapsto_A^* (S', \texttt{halt}\ v)$ *then for some location* ℓ, $v = \texttt{ptr}(\ell)$ *and* $S' = \{\ell \mapsto \texttt{union}_{\langle \mathcal{S}(1)\rangle \cup \langle \mathcal{S}(2)\rangle}(\langle \mathcal{S}(i)\rangle)\}$

3.4 Applications

In this section, I demonstrate via example how alias types can be used, primarily by compiler writers, to verify the safety of low-level intermediate code. First, I show how to verify programs written using the destination-passing style pattern, an efficient technique for constructing recursive data structures. Second, I show how to encode stack typing disciplines in the alias types framework. Finally, I investigate Deutsch-Schorr-Waite or “link-reversal” patterns, which traverse data structures using minimal additional space.

3.4.1 Destination-Passing Style

The *destination-passing style* (DPS) transformation detects certain “almost-tail-recursive” functions and automatically transforms them into efficient properly tail-recursive functions. The transformation improves many functional programs significantly, leading a number researchers

to study the problem in depth [Wad85, Lar89, CO96, Min98]. My contribution is to provide a type system that is capable of verifying that the code resulting from the transformation is safe.

Append is the canonical example of a function suitable for DPS:

```
fun append (xs,ys) =
  case xs of
    [] -> ys
  | hd :: tl -> hd :: append (tl,ys)
```

Here, the second-last operation in the second arm of the case is a function call and the last operation constructs a cons cell. If the two operations were inverted, we would have an efficient tail-recursive function. In DPS, the function allocates a cons cell before the recursive call and passes the partially uninitialized value to the function, which computes its result and fills in the uninitialized part of the data structure.

The following untyped pseudo-code presents append in DPS, making allocation and initialization of the data structures explicit (an empty list is tagged 1 and a non-empty list is tagged 2).

```
fun appendDPS (xs,ys,prev,start,cont) =
  case xs of
    <S(1)> -> free xs; prev.3 := ys; cont (start)
  | <S(2),hd,tl> ->
      let next = new(3) in
      prev.3 := next;
      next.1 := S(2);
      next.2 := hd;
      appendDPS (tl,ys,next,start,cont)
```

In the code above, `xs` and `ys` are as they were before. The node in the list `xs` that was partially processed on the previous iteration is called `prev`. The tail position for `prev` has not yet been filled in, but it will be on this iteration, either by `ys` or by the next node in `xs`. The variable `start` points to the eventual result and `cont` is the continuation. Before calling `appendDPS`, a wrapper function would check for the case that `xs` is initially null, and if not, would allocate the first cell in the output list and pass a pointer to that cell to `appendDPS` as both `prev` and `start`.

If the input list `xs` is linear, it will not be used in the future. In this case, it is possible to further optimize the program by reusing the input list cells for the output list. With this fact in mind, we could rewrite `appendDPS` as follows:

```
fun appendDPS' (xs,ys,prev,start,cont) =
  case xs of
    <S(1)> -> free xs; prev.3 := ys; cont (start)
  | <S(2),hd,tl> -> appendDPS' (tl,ys,xs,start,cont)
```

To facilitate readability, I will define a number of abbreviations before presenting a typed variant of `appendDPS'` in the alias type system. For expository purposes, I will assume the language is augmented with integers and the lists are integer lists. The type of integer lists *List* and their unrolling *List'* is:

$$
\begin{array}{rcl}
List & = & \mu\alpha.\langle\mathcal{S}(1)\rangle \cup \exists[\zeta \mid \{\zeta \mapsto \alpha\}].\langle\mathcal{S}(2), int, ptr(\zeta)\rangle \\
List' & = & \langle\mathcal{S}(1)\rangle \cup \exists[\zeta \mid \{\zeta \mapsto List\}].\langle\mathcal{S}(2), int, ptr(\zeta)\rangle
\end{array}
$$

```
fix append [ϵ, ζxs, ζys, ζp, ζs](ϵ ∗ {ζp ↦ ⟨S(2), int, ptr(ζxs)⟩, ζxs ↦ List, ζys ↦ List},
  xs : ptr(ζxs), ys : ptr(ζys), prev : ptr(ζp), start : ptr(ζs), cont : τc[ϵ, ζp, ζs]).
  unroll(ζxs);
  case xs
    ( inl ⟹
        free xs;                     % 1.
        prev.3:=ys;                  % 2.
        rollList ζp packing ζys;     % 3.
        cont(start)
    | inr ⟹
        unpack ζxs with ζtl ;        % 4.
        let tl = xs.3 in             % 5.
        append[ϵ ∗ {ζp ↦ ⟨S(2), int, ptr(ζxs)⟩}, ζtl, ζys, ζxs, ζs]
          (tl, ys, xs, start, cont′))

where τc[ϵ, ζp, ζs] = (ϵ ∗ {ζp ↦ List}, ptr(ζs)) → 0
```

Figure 3.12: Alias Types: Optimized Append

Given these list definitions, we can define the following composite coercion.

$$\begin{array}{l}\texttt{rollList}\,\zeta_1\,\texttt{packing}\,\zeta_2 = \\ \quad \texttt{pack}_{[\zeta_2|\{\zeta_2\mapsto List\}]\texttt{as}\,\exists[\zeta_2|\{\zeta_2\mapsto List\}].\langle \mathcal{S}(2),int,ptr(\zeta_2)\rangle}(\zeta_1); \\ \quad \texttt{union}_{List'}(\zeta_1); \\ \quad \texttt{roll}_{List}(\zeta_1)\end{array}$$

This coercion operates on a portion of the store with shape

$$\{\zeta_1 \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_2)\rangle\} * \{\zeta_2 \mapsto List\}$$

It packs up ζ_2 into an existential around ζ_1, lifts the resultant object up to a union type and finally rolls it up, producing a store with the shape $\{\zeta_1 \mapsto List\}$.

Figure 3.12 presents the well-typed, optimized function *append*. *append*'s caller passes two pointers to the beginning of the first list (aliases of one another) for parameters *prev* and *start*. It also passes a pointer to the second element in that list for parameter xs and a pointer to the second list for parameter ys. Notice that the contents of location ζ_s are not described by the aliasing constraints. On the first iteration of the loop ζ_s is an alias of ζ_p and on successive iterations, its contents are hidden by ϵ. However, these facts are not explicit in the type structure and therefore ζ_s cannot be used during any iteration of the loop (*cont* will be aware that ζ_s equals ζ_p and may use the resultant list).

The first place to look to understand this code is at the aliasing constraints, which act as a loop invariant. Reading the constraints in the type from left to right reveals that the function expects a store with some unknown part (ϵ) as well as a known part. The known part contains a cons cell at location ζ_p that is linked to a *List* in location ζ_{xs}. Independent of either of these objects is a third location, ζ_{ys}, which also contains a *List*.

The first instruction in the function unrolls the recursive type of the object at ζ_{xs} to reveal that it is a union and can be eliminated by a case statement. In the first branch of the case,

xs must point to null. The code frees the null cell, resulting in a store at program point `1` that can be described by the constraints

$$\epsilon * \{\zeta_p \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{xs})\rangle\} * \{\zeta_{ys} \mapsto List\}$$

Observe that the cons cell at ζ_p contains a dangling pointer to memory location ζ_{xs}, the location that has just been freed and no longer appears in the constraints. Despite the dangling pointer, the code is perfectly safe: The typing rules prevent the pointer from being used.

Next, the second list ys is banged into the cons cell at ζ_p. Hence, at program point `2`, the store has a shape described by

$$\epsilon * \{\zeta_p \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{ys})\rangle\} * \{\zeta_{ys} \mapsto List\}$$

The type of the cons cell at ζ_p is different here than at `1`, reflecting the new link structure of store. The tail of the cell no longer points to location ζ_{xs}, but to ζ_{ys} instead. After packing and rolling using the composite coercion, the store can be described by $\epsilon * \{\zeta_p \mapsto List\}$. This shape equals the shape expected by the continuation (see the definition of τ_c), so the function call is valid.

In the second branch of the case, xs must point to a cons cell. The existential containing the tail of the list is unpacked and at program point `4`, the store has shape

$$\begin{aligned}&\epsilon * \{\zeta_p \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{xs})\rangle\} * \{\zeta_{xs} \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{tl})\rangle\} * \{\zeta_{tl} \mapsto List\} * \\ &\quad \{\zeta_{ys} \mapsto List\}\end{aligned}$$

It is now possible to project the tail of xs. To complete the loop, the code uses polymorphic recursion. At the end of the second branch, the constraint variable ϵ for the next iteration of the loop is instantiated with the current ϵ and the contents of location ζ_p, hiding the previous node in the list. The location variables ζ_{xs} and ζ_p are instantiated to reflect the shift to the next node in the list. The locations ζ_{ys} and ζ_s are invariant around the loop and therefore are instantiated with themselves.

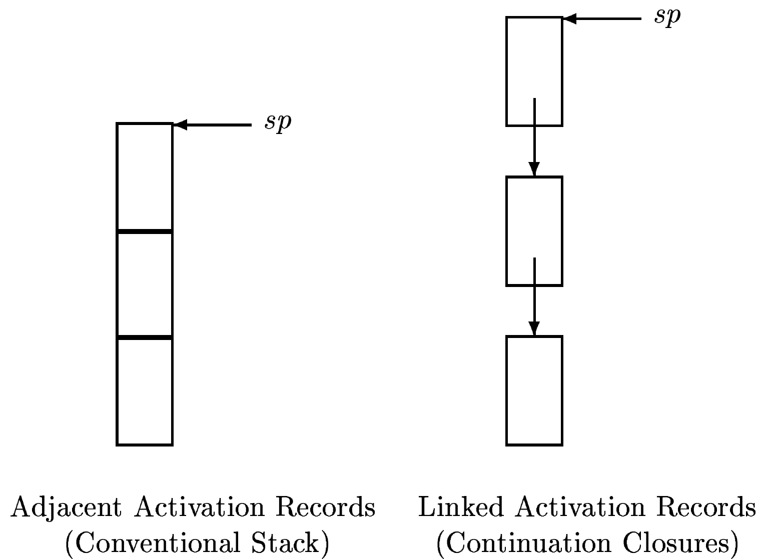
The last problem is how to define the continuation $cont'$ for the next iteration. The function should be tail-recursive, so we would like to use the continuation $cont$. However, close inspection reveals that the next iteration of append requires a continuation with type $\tau_c[\epsilon * \{\zeta_p \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{xs})\rangle\}, \zeta_{xs}, \zeta_s]$ but that the continuation $cont$ has type $\tau_c[\epsilon, \zeta_p, \zeta_s]$. The problem is that this iteration of the recursion has unrolled and unpacked the recursive data structure pointed to by xs, but before "returning" by calling the continuation, the list must be packed and rolled back up again. Therefore, the appropriate definition of $cont'$ is $cont \circ (\texttt{rollList}\, \zeta_p\, \texttt{packing}\, \zeta_{xs})$. Once the continuation packs ζ_{xs} and rolls the contents of location ζ_p into a $List$, the constraints satisfy the requirements of the continuation $cont$. Semantically, $cont'$ is equivalent to the following function.

$$\begin{aligned}&\texttt{fix}\ _\ [\cdot](\epsilon * \{\zeta_p \mapsto \langle \mathcal{S}(2), int, ptr(\zeta_{xs})\rangle\} * \{\zeta_{xs} \mapsto List\}, (start{:}ptr(\zeta_s))).\\ &\quad \texttt{rollList}\, \zeta_p\, \texttt{packing}\, \zeta_{xs};\\ &\quad cont(start)\end{aligned}$$

However, because coercions can be erased before running a program, it is simple to arrange for $cont'$ to be implemented by $cont$.

3.4.2 Stack Typing

In order to compile programming languages with recursive functions, it is necessary to allocate space to save the values of local variables across a function call. Most compilers use a conventional stack to store these values. However, CPS-based compilers, such as the Standard ML of New Jersey [App92], save local variables in the closures of continuation functions. These closures carry the same data as conventional stack-allocated activation records and in the absence of non-local control constructs such as call/cc, they are allocated and used in a first-in, first-out order. The essential difference between the two data structures is that conventional stack-allocated activation records are placed adjacent to one another in memory whereas continuation closures are not usually allocated adjacent to one another. As a result, the callee's continuation closure contains an explicit pointer to link it to its caller's continuation closure:



Adjacent Activation Records (Conventional Stack) Linked Activation Records (Continuation Closures)

When programs allocate and deallocate linked activation records in a stack-like fashion, they can be proven safe using the type system developed in this chapter. For example, assume we want to implement a safe C-like language (*i.e.* a language with no nested function declarations) using a linked list of activation records. If the source-level C function has the type $\tau_1 \rightarrow \tau_2$[8] then a natural type for its implementation is

$$\forall[\epsilon, \zeta_{sp}].(\epsilon, ptr(\zeta_{sp}), \tau_1', \tau_{cont}) \rightarrow \mathbf{0}$$

where

ϵ	$=$	the stack
ζ_{sp}	$=$	location of the caller's activation record
τ_{cont}	$=$	$(\epsilon, ptr(\zeta_{sp}), \tau_2') \rightarrow \mathbf{0}$
τ_1'	$=$	implementation type corresponding to τ_1
τ_2'	$=$	implementation type corresponding to τ_2

There are now three arguments to the function: a stack pointer (with type $ptr(\zeta_{sp})$), an argument (with type τ_1') and a continuation or return address (with type τ_{cont}). In a C-like

[8]For simplicity, I will assume τ_1 and τ_2 are the types of small values so their translation need add nothing to the store type.

language, functions only have access to explicit parameters, their own local variables and global variables (which are straightforward to add, but will not be modeled here). They do not, in general, have access to the locals of their callers. The store variable ϵ hides the contents of the caller's activation record and all other preceding activation records from the callee, thereby restricting the callee's access to its own locals until the continuation is invoked.

When a function is called, it allocates an activation record of sufficiently large size to hold its locals and to store the results of intermediate computations. This process is identical to the construction of any other data structure. One slot in the activation record should be reserved for the continuation (return address) and another slot should be reserved for the pointer to its caller:

fix $f\ [\epsilon, \zeta_{sp}](\epsilon, sp{:}ptr(\zeta_{sp}), arg{:}\tau_1', cont{:}\tau_{cont}).$
    let $\zeta_{sp}', sp' =$ new (n) in   `%` $n =$ `size of activation record`
    $sp'.n{:=}sp;$ `% store pointer to caller's act. record`
    $sp'.(n-1){:=}cont;$ `% store pointer to return address`
    $\vdots$

At a function call site, the caller passes a pointer to its activation record, an argument of the correct type and a continuation to the callee. If the function above (f) has only a single value (the integer x) that needs to be saved across a function call, then the activation record for f should have the type $\langle int, \tau_{cont}, ptr(\zeta_{sp})\rangle$. In this case, a recursive call of f to itself has the form:

    $\vdots$
    $sp'.1{:=}x;$ `% store` x `in activation record`
                `% store type =` $\epsilon * \{\zeta_{sp}' \mapsto \langle int, \tau_{cont}, ptr(\zeta_{sp})\rangle\}$
  $f[\epsilon * \{\zeta_{sp}' \mapsto \langle int, \tau_{cont}, ptr(\zeta_{sp})\rangle\}, \zeta_{sp}'](sp', arg', cont')$

As in the previous section, I use polymorphic recursion to establish the safety of a recursive function call. The callee's type parameter ϵ_{callee} is instantiated with the stack ϵ joined with a type for the caller's activation record: $\{\zeta_{sp}' \mapsto \langle int, \tau_{cont}, ptr(\zeta_{sp})\rangle\}$.

If a compiler uses these implementation techniques then there will always be exactly one pointer to any activation record: The pointer to the topmost activation record resides in one of the current function's local variable (sp) and every other activation record in the stack is pointed to by the activation record of the function it called. Since activation records are unshared, linear types and polymorphism are sufficient to give types to these stacks; the more advanced features presented in this chapter are unnecessary for this particular compilation strategy. However, implementation techniques for other language features, particularly Pascal-like nested functions and (in some cases) exceptions, create aliasing amongst activation records.

Consider a language that allows nested but not first-class functions.[9] Since nested functions may contain references to the local variables of their enclosing functions, there must be some way to access the activation records of enclosing functions. One way to implement such a language is to use a display, which is simply a data structure that points to the activation records of the lexically enclosing functions of the current function (See Aho, Sethi and Ullman [ASU86] for details). Displays create aliasing because activation records are pointed to both by a dynamic

[9] Functions may not be stored in data structures or returned as results. Pascal allows function parameters, but I do not consider them here either.

link that, as in the simple stack model, connects caller to callee, and also by a static link that connects a function to its statically enclosing functions.

The store typing invariant for a display with two lexical levels can be described by the following type:

$$\begin{array}{l}\{\zeta_{lex2} \mapsto \langle \ldots, ptr(\zeta_{lex2caller})\rangle\}* \\ \{\zeta_{lex1} \mapsto \langle \ldots, ptr(\zeta_{lex1caller})\rangle\}* \\ \{\zeta_{display} \mapsto \langle ptr(lex1), ptr(lex2)\rangle\}* \\ \epsilon\end{array}$$

At any given program point, one program variable (sp with type $ptr(\zeta_{lex2})$) points to the top of the stack and another variable ($display$ with type $ptr(\zeta_{display})$) points to the display. The location ζ_{lex2} holds the activation record for the function currently being executed and $\zeta_{\zeta lex2caller}$ is the location storing the activation record of its caller. The pointer with type $ptr(\zeta_{lex2caller})$ is the dynamic link to its caller. The activation record of an enclosing function at lexical depth 1 is stored in location ζ_{lex2} and its caller's activation record is stored in $\zeta_{lex1caller}$.

3.4.3 Deutsch-Schorr-Waite Algorithms

Deutsch-Schorr-Waite or "link reversal" algorithms, are well-known algorithms for traversing data structures while incurring minimal additional space overhead. These algorithms were first developed for executing the mark phase of a garbage collector [SW67]. During garbage collection, there is little or no extra space available for storing control information, so minimizing the overhead of the traversal is a must. Recent work by Sobel and Friedman [SF98] has shown how to automatically transform certain continuation-passing style programs, those generated by *anamorphisms* [MFP91], into link-reversal algorithms. Here I give an example how to encode a link-reversal algorithm in our calculus.

For this application, I will use the definition of trees from section 3.1.

$$\begin{array}{l}Tree = \\ \quad \mu\alpha.\langle \mathcal{S}(1)\rangle \cup \exists[\zeta_L, \zeta_R \mid \{\zeta_L \mapsto \alpha, \zeta_R \mapsto \alpha\}].\langle \mathcal{S}(2), ptr(\zeta_L), ptr(\zeta_R)\rangle\end{array}$$

$$\begin{array}{l}Tree' = \\ \quad \langle \mathcal{S}(1)\rangle \cup \exists[\zeta_L, \zeta_R \mid \{\zeta_L \mapsto Tree, \zeta_R \mapsto Tree\}].\langle \mathcal{S}(2), ptr(\zeta_L), ptr(\zeta_R)\rangle\end{array}$$

The code for the algorithm appears in figures 3.13 and 3.14. The trick to the algorithm is that when recursing into the left subtree, it uses space normally reserved for a pointer to that subtree to point back to the parent node. Similarly, when recursing into the right subtree, it uses the space for the right pointer. In both cases, it uses the tag field of the data structure to store a continuation that knows what to do next (recurse into right subtree or follow the parent pointers back up the tree). Before ascending back up out of the tree, the algorithm restores the link structure to a proper tree shape and the type system checks this is done properly. Notice that all of the functions and continuations are closed, so there is no stack hiding in the closures.

3.5 Discussion

Reasoning about aliasing is an extremely important and well-researched topic as pointers and sharing are ubiquitous in all mainstream programming languages. Surprisingly, there has been very little work on combining aliasing information with type systems for high-order languages

```
% Traverse a tree node
letrec walk [ε, ρ1, ρ2](ε * {ρ1 ↦ Tree}, t : ptr(ρ1), up : ptr(ρ2), cont : τc[ε, ρ1, ρ2]).
   unroll(ρ1);
   case t of
      ( inl ⟹
         union_Tree'(ρ1);
         roll_Tree(ρ1);
         cont(t, up)
      | inr ⟹
         unpack ρ1 with ρL, ρR ;
         %  store cont in tag position
         t.1:=cont;
         let left = t.2 in
         %  store parent pointer as left subtree
         t.2:=up;
         walk[ε * {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρ2), ptr(ρR)⟩} * {ρR ↦ Tree}, ρL, ρ1]
            (left, t, rwalk[ε, ρ1, ρ2, ρL, ρR]))

% Walk the right-hand subtree
and rwalk [ε, ρ1, ρ2, ρL, ρR]
      (ε * {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρ2), ptr(ρR)⟩} * {ρL ↦ Tree} * {ρR ↦ Tree},
      left : ptr(ρL), t : ptr(ρ1)).
   let up = t.2 in
   % restore left subtree
   t.2:=left;
   let right = t.3 in
   %  store parent pointer as right subtree
   t.3:=up;
   walk[ε*
         {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρL), ptr(ρ2)⟩}*
         {ρL ↦ Tree}, ρR, ρ1]
      (right, t, finish[ε, ρ1, ρ2, ρL, ρR])
```

where $\tau_c[\epsilon, \rho_1, \rho_2] = (\epsilon * \{\rho_1 \mapsto Tree\}, ptr(\rho_1), ptr(\rho_2)) \rightarrow \mathbf{0}$

Figure 3.13: Deutsch-Schorr-Waite Tree Traversal

```
% Reconstruct tree node and return
and finish [ε, ρ1, ρ2, ρL, ρR]
        (ε ∗ {ρ1 ↦ ⟨τc[ε, ρ1, ρ2], ptr(ρL), ptr(ρ2)⟩} ∗ {ρL ↦ Tree} ∗ {ρR ↦ Tree},
        right : ptr(ρR), t : ptr(ρ1)).
    let up = t.3 in
    % restore right subtree
    t.3:=right;
    let cont = t.1 in
    % restore tag
    t.1:=S(2);
    pack_{ρL,ρR}(ρ1);
    union_{Tree'}(ρ1);
    roll_{Tree}(ρ1);
    cont(t, up)
```

where $\tau_c[\epsilon, \rho_1, \rho_2] = (\epsilon * \{\rho_1 \mapsto Tree\}, ptr(\rho_1), ptr(\rho_2)) \rightarrow \mathbf{0}$

Figure 3.14: Deutsch-Schorr-Waite Tree Traversal, Cont.

as I have done here. There is definitely more work to be done in this area to determine both the limitations and practical possibilities of this research.

3.5.1 Arrays

Arrays involve non-trivial extensions to the language described in this chapter. If we attempt to treat arrays in the same way that we treat tuples then different elements of an array will have different types at various points in a computation. In this case, the type system will have to represent array types as dependent functions from array indices to types and in order to decide the type of an array projection operation, the type system will have to be equipped with a decision procedure for (some fragment of) arithmetic. It may be possible to adapt Xi and Pfenning's work on Dependent ML [XP98, Xi99] to accomplish this task, but this path would complicate (an already complex) type system significantly.

A simpler, more practical alternative is to diverge from the treatment of arrays as tuples and to design language constructs that preserve the types of array elements from one operational step to the next, yet retain the strong memory management properties of the language. This design path immediately forces us to define constructs for atomic allocation and initialization of arrays, as in conventional typed languages. Element-wise initialization would cause array elements to take on two different types (uninitialized junk and valid element types) and requires the complex dependent type structure discussed above.

Array projection and update operations could be handled using an atomic swap operation:

$$\texttt{let}\ \zeta_z, z = \texttt{swap}(a, v_1, v_2)\ \texttt{in}\ e$$

Operationally, the swap expression projects the element z from array a at offset v_1 and inserts object v_2 at the same offset. The elements of an array implicitly have existential type, just as the elements of other aggregate data structures such as lists and trees were existentials. The

existential abstracts the names of the locations used to store each array element. The `swap` operation packs the storage required for v_2 and unpacks the storage required for z, making the location for z (*i.e.*, ζ_z) accessible. The typing rule for `swap` follows.[10]

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_A a : ptr(\zeta_a) \\ \Delta; \Gamma \vdash_A v_1 : int \\ \Delta; \Gamma \vdash_A v_2 : ptr(\zeta_2) \\ \Delta \vdash_A C = C' * \{\zeta_a \mapsto \tau\,\texttt{array}\} * \{\zeta_2 \mapsto \tau\} : \texttt{Store} \\ \Delta, \zeta_z{:}\texttt{Loc}; \Gamma, z{:}ptr(\zeta_z); C' * \{\zeta_a \mapsto \tau\,\texttt{array}\} * \{\zeta_z \mapsto \tau\} \vdash_A e\end{array}}{\Delta; \Gamma; C \vdash_A \texttt{let}\,\zeta_z, z = \texttt{swap}(a, v_1, v_2)\,\texttt{in}\,e}$$

The swap operation should ensure that objects in arrays are unshared. Therefore they can be extracted from an array at any time and safely deallocated. Of course, whenever an object is projected from an array, another is put in its place and therefore it may seem as though we cannot do any effective memory management with arrays in this language. However, each array element can be given an option type, either nil or the element itself. In this case, to deallocate an object contained in an array, we simply swap in nil, swap out the legitimate object and free it. The cost of this programming technique is that we must check for nil before using array elements.

3.5.2 Related Work

This research has much in common with efforts to define program logics for reasoning about aliasing [Bur72, CO75, Möl93, Rey00, IO00]. In particular, if we view propositions as types, there are striking similarities with recent work by Reynolds [Rey00] who builds on earlier research by Burstall [Bur72]. Reynolds' logic employs a "spatial conjunction" operator which, like linear logic's tensor, is not contractive. This spatial conjunction separates *propositions* that depend upon disjoint portions of the store similarly to the way that the $*$ operator separates *types* for disjoint portions of the store. Using the spatial conjunction, Reynolds defines Hoare logic rules for an imperative programming language such that a certain degree of local reasoning is possible: Unlike standard Hoare-logic rules for assignment that perform a global substitution, Reynolds formulation ensures that assignment alters at most one of the propositions joined by the spatial conjunction.

Ishtiaq and O'Hearn [IO00] have further analyzed Reynolds' rules in the context of the logic of bunched implications. They are able to give both an intuitionistic and a classical interpretation of the logic (Reynolds could only give an intuitionistic semantics) and to introduce operations for safe object deallocation. Moreover, they have reformulated Reynolds' Hoare rules and proven that they generate weakest preconditions.

The Hoare rules defined by Reynolds, Ishtiaq and O'Hearn are highly similar to the typing rules defined in this chapter. The most significant difference between the two pieces of work is that Reynolds, Ishtiaq and O'Hearn use a full first-order logic including implication, conjunction, disjunction, equalities and first-order quantifiers to express properties of the store whereas my type system is relatively inexpressive. On the other hand, it is possible for a mechanical type checker to verify our code, without requiring human intervention or a sophisticated theorem prover. In fact, the central elements of the type system have been implemented in the context of Cornell's typed assembly language project [TAL].

[10] A slightly modified rule is necessary to deal with arrays of small values such as integers.

One way to approach the expressiveness of Hoare logic in a type system may be to combine alias types with the dependent type system devised by Xi and Pfenning [XP99, Xi99]. They combine singleton types with logical and arithmetic constraints to tackle problems such as array bounds check elimination [XP98]. They are also able to specify and check the correctness of small programs including red-black tree algorithms and an interpreter for the simply-typed lambda calculus. My choice to use singleton types and polymorphism rather than more traditional dependent type theory was largely inspired by their research.

There are also similarities with alias analysis techniques for imperative languages [JM81, LH88, Deu94, GH96, SRW98]. Alias types appear most closely related to the shape analysis developed by Sagiv, Reps, and Wilhelm (SRW) [SRW98], which has also been used to develop sophisticated pointer logics [SRW99, BRS99]. Although the precise relationship between systems is unclear, several of the key features that make SRW shape analysis more effective than similar alias analyses can be expressed in my type system. More specifically:

1. Unlike some other analyses, SRW shape nodes do not contain information about concrete locations or the site where the node was allocated. My type system drops information about concrete locations using location polymorphism.

2. SRW shape nodes are named with the set of program variables that point to that node. My type system can only label a node with a single name, but it is able to express the fact that a set of program variables point to that node using the same singleton type for each program variable in the set.

3. SRW shape nodes may be flagged as unshared. Linear types account for unshared shape nodes.

4. A single SRW summary node describes many memory blocks, but through the process of *materialization* a summary node may split off a new, separate shape node. Some summary nodes may be represented as recursive types in my framework and materialization can be explained by the process of unrolling and unpacking a recursive and existential type.

One of the advantages of my approach is that my language makes it straightforward to create dependencies between functions and data using store or location polymorphism. For example, in my implementation of the Deutsch-Schorr-Waite algorithm, I manipulate continuations that know how to reconstruct a well-formed tree from the current heap structure and I am able to express this dependence in the type system. Explicit manipulation of continuations is necessary in sufficiently low-level typed languages such as Typed Assembly Language when return addresses are interpreted as continuations [MCGW98].

Several other authors have considered alternatives to pure linear type systems that increase their flexibility. For example, Kobayashi [Kob99] extends standard linear types with data-flow information and Minamide [Min98] uses a linear type discipline to allow programmers to manipulate "data structures with a hole." Minamide's language allows users to write programs that are compiled into destination-passing style. However, Minamide's language is still quite high-level; he does not show how to verify explicit pointer manipulation. Moreover, neither of these type systems provide the ability to represent cyclic data structures.

3.5.3 Limitations

The type system described in this chapter is considerably more flexible than a linear type system, but it is also considerably more complex. Hence, before adapting this technology, a

programmer must carefully consider whether their application requires the extra expressiveness or if a simpler solution will do. On the other hand, the type system is not without its limitations. For some applications, it may be necessary to move to a proof-carrying code framework and employ a more powerful logic along the lines proposed by Reynolds or Ishtiaq and O'Hearn.

One significant limitation of the type system is the inability to represent storage blocks that may (or may not) alias one another. In many contexts, precise must-alias information is not necessary to ensure code safety and using imprecise may-or-may-not-alias information can result in better code reuse. For example, consider an `add` function that sums the contents of its arguments. In an untyped language, we might write:

```
λ(x, y, cont).
let t1 = x.1 in              % project first integer
let t2 = y.1 in              % project second integer
let result = t1 + t2 in      % add them together
cont(result)                 % return result
```

This code is safe regardless of whether x and y point to the same memory block or not and as a result, it is not clear whether `add` should have the type

$$\forall[\epsilon, \zeta_1, \zeta_2].(\epsilon * \{\zeta_1 \mapsto \langle int \rangle\} * \{\zeta_2 \mapsto \langle int \rangle\}, ptr(\zeta_1), ptr(\zeta_2), \tau_{cont}) \rightarrow \mathbf{0}$$

or

$$\forall[\epsilon, \zeta_1].(\epsilon * \{\zeta_1 \mapsto \langle int \rangle\}, ptr(\zeta_1), ptr(\zeta_1), \tau_{cont}) \rightarrow \mathbf{0}$$

Each of these two types is appropriate for different contexts; neither one is better than the other. More generally, any n-ary non-destructive function[11], where n is greater than one, is safe regardless of aliasing structure and yet cannot be efficiently encoded in this language. Either the code for these functions, or, in certain circumstances, their arguments, must be copied to satisfy the type system. In chapter four, I will study this problem in more detail and provide a solution.

A more general problem occurs when some data structure has (statically) unknown aliasing structure. For instance, it is not possible to define or manipulate a general directed graph in the language of alias types. Assuming a node in this graph has two outgoing edges, then we could give the corresponding memory block the type $\langle ptr(\zeta_1), ptr(\zeta_2) \rangle$. However, to use the links in the node, we must know *exactly* which two other nodes $ptr(\zeta_1)$ and $ptr(\zeta_2)$ point to. In the case of a general graph, we do not have this information. Chapter four also provides a solution to this problem, based on Tofte and Talpin's notion of memory regions [TT94].

Unfortunately, there are also some data structures that have regular structure but that cannot be handled effectively using the techniques developed here or in future chapters. One problem is that the logic describing the store has a very specialized form. It is essentially a conjunction of points-to constraints that describe the contents of individual locations. This specialized form makes the type system easier to decide but restricts its expressiveness. There are many memory structures that can be summarized if a full and unconstrained logic is used to describe the shape of the store that my restricted logic cannot handle. For example, this formula:

$$(x = y \Rightarrow C_1) \wedge (x \neq y \Rightarrow C_2)$$

[11] A non-destructive function is one that does not alter the state of its arguments either by deallocation or assignment.

implies that the shape of the store (either C_1 or C_2) depends upon the relationship between variables x and y. Such constraints cannot be expressed in my language. Of course, the cost of such expressiveness is that it is necessary to use a theorem prover (bound to be incomplete if the logic is powerful enough) or human intervention (time-consuming and therefore expensive) to decide the validity of these formulae.

The last main limitation of this type system is that the introduction and elimination rules, combined with a lack of equalities between types, force data structures within the store to take on a specialized form. In particular, unions always have the form

$$\begin{array}{c} \exists[\Delta_1 \mid C_1].\cdots\exists[\Delta_j \mid C_j].\langle \mathcal{S}(1), \tau_1, \ldots, \tau_k \rangle \\ \cup \\ \exists[\Delta'_1 \mid C'_1].\cdots\exists[\Delta'_l \mid C'_l].\langle \mathcal{S}(2), \tau'_1, \ldots, \tau'_m \rangle \end{array}$$

and the only way to eliminate them is by testing the singleton tags. There are several situations in which it is desirable to have more flexibility. For instance, it is possible to define another elimination form that tests equality of the union against a pointer of known type instead of testing the tags. Using this mechanism, a data structure could save one memory word per union. It is also useful to be able to be able to descriminate between pointers and small integers (zero, for example, to represent null). Using this representation, the definition of a list type is:

$$\mu\alpha.\mathcal{S}(0) \cup \exists[\zeta \mid \{\zeta \mapsto \alpha\}].\langle ptr(\zeta) \rangle$$

Notice that the union takes on a slightly different form here; the first alternative is a simple singleton. Such a union would require a new introduction rule as well as a new elimination rule.

Yet another variant arises when implementing a queue. Assuming the head of the queue points to ζ_1 (variable *head* has type $ptr(\zeta_1)$) and the tail of the queue points to ζ_2 (variable *tail* has type $ptr(\zeta_2)$) then the queue's body can be described using these constraints:

$$\{\zeta_1 \mapsto \mu\alpha.ptr(\zeta_2) \cup \exists[\zeta_3 \mid \{\zeta_3 \mapsto \alpha\}].\langle ptr(\zeta_3) \rangle\} * \{\zeta_2 \mapsto \langle \mathcal{S}(0) \rangle\}$$

Here the union takes on yet another form and new coercions are necessary to construct and manipulate the queue.

The central point of these examples is that although they fall into the general framework provided by the language, each new data invariant requires additional *ad hoc* typing rules and coercions. These new rules complicate the implementation and pollute the trusted computing base. Therefore, in summary, alias types are an effective mechanism for tracking local aliasing and can model stacks as well as certain forms of lists and trees, but to handle more complicated data structures, it may be necessary or at least more effective either to move to a more general purpose logic or to coarsen the granularity at which memory management is performed — the latter approach being the central topic of the next chapter.

Chapter 4

Regions

Double, double, toil and trouble;
Fire burn, and cauldron bubble.

– *William Shakespeare. Macbeth.*

In real programs, the aliasing relationships between individual objects are often too complex for compiler-writers or programmers to keep track of. One technique for handling these situations is to give up on tracking the references to every object precisely and instead to group objects with similar lifetimes into one of many *memory regions*. This design simplifies the memory management problem as a region-based system need only track region aliases, which are fewer and have less complex structure than per-object aliases.

Tofte and Talpin [TT94, TT97] realized the benefits of region-based memory management and recognized that it might be possible to infer region allocation and deallocation points for ML programs automatically. They developed a sound, non-deterministic collection of inference rules (not yet an algorithm), inspired by earlier work on type and effects systems by Gifford, Lucassen, Jouvelot and Talpin [GL86, JG91, TJ92]. In later work, Tofte and Birkedal [TB98] developed a remarkably effective inference algorithm and proved it sound and terminating, though incomplete. Tofte, with others, has implemented the algorithm and developed many sophisticated optimization techniques [BTV96]. Currently, Tofte's implementation, the ML Kit with Regions [TBE$^+$98], is competetive with, if not superior to, other compilers for ML in terms of time and space on many memory-intensive programs.

In this chapter, I will develop a new type system for region-based memory management. This research is based on joint work I have done with Karl Crary and Greg Morrisett [CWM99, WCM00]. The main innovation of the work is that it combines ideas from linear type systems with Tofte and Talpin's type and effect system. An important contribution of this language design is that it gives rise to a more flexible region deallocation principle than Tofte and Talpin's calculus, yet it can be proven sound using Wright and Felleisen's Subject Reduction and Progress properties.

In the next section, I will review the highlights of the Tofte-Talpin calculus to show where it lacks expressiveness. Section 4.2 develops the technical aspects of the new region-based language, which I call the "Capability Calculus" because it is based on the capability mechanism of the previous chapter.[1] Section 4.5 informally discusses different ways of handling recursive

[1] Historically, the region-based capability calculus was developed first (see [CWM99]) and the alias types of the previous chapter were derived from it (see [SWM00, WM00]).

data types in the language and the advantages of mixing alias types with regions. It also comments on related work.

4.1 Introduction to Region-based Memory Management

In order to ensure that regions are used safely, the Tofte-Talpin language includes a lexically-scoped expression (`letregion r in e end`) that delimits the lifetime of a region `r`. A region is allocated when control enters the scope of the `letregion` construct and is deallocated when control leaves the scope. Programs may allocate values into live regions using the notation `v at r`. These values may be used until the region is deallocated. For example,

```
                   ⋮
                 ⎧ letregion r in         % Allocate region r
Region lifetime  ⎨     let x = v at r in  % Allocate value v in r
                 ⎪     f (r,x)            % Call f, may access r
                 ⎩ end                    % Deallocate r (and v)
                   ⋮
```

Tofte and Talpin ensure that deallocated values are not accessed unsafely using a type and effects system. Informally, whenever an expression uses a value in region `r`, the type system expresses this fact using the effect *access*(`r`). However, effects occuring within the scope of the `letregion` construct are masked. More specifically, if the expression `e` has effects *access*(`r`) $\cup\ \psi$ (for some set of effects ψ) then the overall effect of the expression `letregion r in e end` is simply ψ. Hence, if there is no overall effect for an entire program then every region access must have occured within the scope of the corresponding `letregion` construct. In other words, values in region `r` are used only during the lifetime of `r` and not before or after. If this condition holds, we can conclude the program is safe.

The Tofte-Talpin language makes efficient use of memory provided that the lifetimes of values coincide with the lexical structure of the program. However, if lifetimes deviate from program structure then this style of region-based memory management may force programs to use considerably more memory than necessary. Consider the following (yet to be region-annotated) program fragments.

```
% Scope 1:  The Call Site

let x = v in
⋮
let y = f (x) in
⋮
y is dead
```

```
% Scope 2:  The Function

fun f (x) =
    ⋮
    x is dead
    ⋮
    let y = v′ in
    ⋮
    return y
```

The value `v` is an argument to the function `f` and must be allocated in the scope of the function call. However, when `f` is executed, `v` dies quickly. The value `v′` exhibits the inverse behaviour. It is allocated inside `f` but is returned as the function result. Both `v` and `v′` have lifetimes that

span two lexical scopes, but neither is live for very long in either scope. Consequently, vanilla region inference does not perform well in this setting. The best it can do is wrap the function call in a pair of `letregion` commands.

```
% Scope 1:  The Call Site

letregion r in
    let x = v at r in
    ⋮
    letregion r′ in
        let y = f (r,r′,x) in
        ⋮
        y is dead
    end (r′)
end (r)
⋮
```

```
% Scope 2:  The Function

fun f (r,r′,x) =
    ⋮
    x is dead
    ⋮
    let y = v′ at r′ in
    ⋮
    return y
```

Here, the regions `r` and `r′` are live much longer than they need to be due to the inflexibility of the `letregion` construct. Both regions must be allocated outside the function call. Notice also that even though `v` is dead when the function call returns, the outer region `r` cannot be deallocated until after the inner region `r′` has been deallocated. Lexical scoping enforces a stack-like, last-allocated/first-deallocated memory management discipline.

In this example, a much better solution is to provide two separate commands for region allocation (`newregion`) and region deallocation (`freeregion`). The following program takes this approach. In principle, since the lifetimes of regions `r` and `r′` do not overlap, the memory for these regions could be reused.

```
% Scope 1:  The Call Site

let newregion r in
let x = v at r in
⋮
let r′,y = f (r,x) in
⋮
y is dead
let freeregion r′ in
⋮
```

```
% Scope 2:  The Function

fun f (r,x) =
    ⋮
    x is dead
    let freeregion r in
    ⋮
    let newregion r′ in
    let y = v′ at r′ in
    ⋮
    return (r′,y)
```

Unfortunately, we cannot write this program in the Tofte-Talpin language because it is based on the idea of lexical scoping. Another consequence of this language design is that any transformation that alters program structure can affect memory management. One of the most devastating transformations for the Tofte-Talpin type system is the continuation-passing style transformation. CPS places each successive computation in the scope of all previous computations, with the result that no regions can be deallocated until the entire computation has been completed. In the following example, the CPS transformation prevents the region `r`

from being deallocated until after `code` has been executed when it could be deallocated as soon as `f` has completed its computation.

```
letregion r in                       letregion r in
    f (r,v)              ⇒               f (r,v,λ.code)
end;                                 end
code
```

The observation that the Tofte-Talpin type system will make poor use of memory in such cases has been made before. Both Birkedal *et al.* [BTV96] and Aiken *et al.* [AFL95] have proposed optimizations that allow regions to be freed early. However, although their optimizations are safe, there is no simple proof- or type-checker that an untrusting client can use to check the output code. Therefore, in order to construct a verifyably safe, efficient region-based language, we must rethink the language design.

4.2 The Capability Calculus

The Capability Calculus is a new statically-typed intermediate language that supports the explicit allocation, freeing and accessing of memory regions. The key aspect of its design is the incorporation of notions of linear and non-linear regions. Linear regions make it possible to define a new principle for region deallocation and result in a type system that is strictly more powerful than Tofte and Talpin's type system.

This section defines the syntax and static semantics of the Capability Calculus. Like the language of the previous chapter, Capability Calculus are written in continuation-passing style. The complete syntax of the language appears in Figure 4.1. I will discuss the semantics of the language in the following subsections.

4.2.1 Operational Semantics

We specify the operational behavior of the Capability Calculus using a call-by-value allocation semantics [MFH95, MH97], which makes the allocation of data in memory explicit. The semantics, which is specified in Figure 4.2, is given by a deterministic rewriting system $P \longmapsto_R P'$ mapping programs to programs. As before, a program consists of a pair (S, e) of a store and an expression to be executed.

The store now has a slightly more complicated structure. It is represented as a finite mapping of *region names* (ν) to *memory regions* where a memory region itself is a finite mapping from locations to stored values. When convenient, I abbreviate $S(\nu)(\ell)$ by $S(\nu.\ell)$ and $S\{\nu \mapsto S(\nu)\{\ell \mapsto h\}\}$ by $S\{\nu.\ell \mapsto h\}$.

Regions are created at run time by the declaration **newrgn** ρ, x, which extends the store with a new region (ν)[2], substitutes ν for ρ in the following instructions, and the *handle* for the region ($\mathtt{handle}(\nu)$) for x.[3] Notice the strong similarity between region variables ρ and location variables ζ and between region handles and pointers. The former are compile-time concepts that are used during type checking. The latter are data structures needed at run time to access the appropriate memory structure (either a region or a simple memory block). More specifically,

[2]A "new" region is one that does not occur anywhere in the current memory (*i.e.*, the region's name does not occur in the domain of current memory nor does it occur in any stored value) or in the expression being executed.

[3]Both ρ and x are considered bound variables for the purposes of alpha-conversion.

kinds	κ	$::=$	$\texttt{Type} \mid \texttt{Rgn} \mid \texttt{Store}$
constructor vars	α, ρ, ϵ		
constructors	c	$::=$	$\alpha \mid \tau \mid r \mid C$
types	τ	$::=$	$\alpha \mid int \mid handle(r) \mid \langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } r \mid$
			$\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \texttt{ at } r$
regions	r	$::=$	$\rho \mid \nu$
capabilities	C	$::=$	$\epsilon \mid \emptyset \mid \{r\} \mid C_1 * C_2 \mid \overline{C}$
con. contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha{:}\kappa \mid \Delta, \epsilon \leq C$
value contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}\tau$
region types	Υ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
memory types	Ψ	$::=$	$\{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}$
small val's	v	$::=$	$x \mid i \mid \nu.\ell \mid \texttt{handle}(\nu) \mid v[c]$
storable val's	h	$::=$	$\texttt{fix} f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid \langle v_1, \ldots, v_n \rangle$
arithmetic ops	p	$::=$	$+ \mid - \mid \times$
declarations	d	$::=$	$x = v \mid x = v_1 \; p \; v_2 \mid x = h \texttt{ at } v \mid x = \pi_i v \mid$
			$\texttt{newrgn } \rho, x \mid \texttt{freergn } v$
terms	e	$::=$	$\texttt{let } d \texttt{ in } e \mid \texttt{if } v \; (e_2 \mid e_3) \mid v(v_1, \ldots, v_n) \mid \texttt{halt } v$
memory regions	R	$::=$	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
stores	S	$::=$	$\{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\}$
programs	P	$::=$	(S, e)

Figure 4.1: Capabilities: Syntax

region handles are needed when allocating objects within a region and when freeing a region, but not when reading data from a region. In the last case, only an object pointer is required. Region handles are normally implemented as a pair consisting of a pointer to the beginning of the region and pointer to the next available (*i.e.* unallocated) address in the region. However, the internal structure of region handles is unimportant for our purposes. They may be thought of simply as pointers to regions.[4]

Regions are freed by the declaration `freergn` v, where v is the handle for the region to be freed. Objects h large enough to require heap allocation (in this language, functions and tuples) are allocated by the declaration $x = h\,\texttt{at}\,v$, where v is the handle for the region in which h is to be allocated. Data are read from a region in two ways: functions are read by a function call, and tuples are read by the declaration $x = \pi_i(v)$, which binds x to the data residing in the ith field of the object at address v. Each of these operations may be performed only when the region in question has not already been freed.

A region maps locations (ℓ) to heap values. Thus, an address is given by a pair $\nu.\ell$ of a region name and a location. In the course of execution, word-sized values (v) will be substituted for value variables and type constructors for constructor variables, but heap values (h) are always allocated in the store and referenced indirectly through an address. Thus, when executing the declaration $x = h\;\texttt{at}\;v$ (where v is $\texttt{handle}(\nu)$, the handle for region ν), h is allocated in region ν (say at ℓ) and the address $\nu.\ell$ is substituted for x in the following code.

A term in the Capability Calculus consists of a series of declarations ending in either a conditional expressioin, a function call, or a halt expression. The class of declarations includes those constructs discussed above, plus two standard constructs, $x = v$ for binding variables to values and $x = v_1\;p\;v_2$ (where p ranges over $+$, $-$ and $\times$) for integer arithmetic.

For example, the program below allocates a region and puts a pair of integers inside it. Next, the components of the pair are projected from the tuple and the region is deallocated. Finally, the program sums the two integers and terminates.

```
let newrgn ρ, x_ρ       in    % Allocate region ρ
let y = ⟨1,2⟩ at x_ρ    in    % Allocate pair in ρ
let t_1 = π_1 y         in    % Access region ρ, no handle required
let t_2 = π_2 y         in
let freergn (x_ρ)       in    % Deallocate region ρ
let z = t_1 + t_2       in
halt z                        % Terminate
```

4.2.2 Types

The types of the Capability Calculus include type constructor variables and *int*, a type of region handles, as well as tuple and function types. If r is a region (*i.e.* r is either a region name ν or, more frequently, a region variable ρ), then $handle(r)$ is the type of r's region handle. This is a *singleton type*, just like the pointer types of the previous chapter. Hence, as before, we can use this type to track certain facts about aliasing. If two values, v and v', both have the same type $handle(r)$ then they are handles for the same region. I also use singleton types to connect objects with the region they inhabit. For example, if a tuple is allocated using a handle with singleton type $handle(r)$, then the tuple has type $\langle\tau_1,\ldots,\tau_n\rangle\;\texttt{at}\;r$. The function type $(C,\tau_1,\ldots,\tau_n)\to\mathbf{0}\;\texttt{at}\;r$ contains functions taking n arguments (with types τ_1 through τ_n)

[4] The interested reader may refer to the ML Kit with Regions [TBE+98] for more details.

$(S, e) \longmapsto_R (S', e')$

$$(S, \texttt{let}\, x = v \,\texttt{in}\, e') \longmapsto_R (S, e'[v/x])$$

$$(S, \texttt{let}\, x = i\ p\ j \,\texttt{in}\, e') \longmapsto_R (S, e'[k/x])$$
where $k = i\ p\ j$

$$(S, \texttt{let}\, x = h\ \texttt{at}\ (\texttt{handle}(\nu)) \,\texttt{in}\, e') \longmapsto_R (S\{\nu.\ell \mapsto h\}, e'[\nu.\ell/x])$$
where $\nu \in Dom(S)$ and $\ell \notin Dom(S(\nu))$

$$(S, \texttt{let}\, x = \pi_i(\nu.\ell) \,\texttt{in}\, e') \longmapsto_R (S, e'[v_i/x])$$
where $\nu \in Dom(S)$ and $\ell \in Dom(S(\nu))$ and $S(\nu.\ell) = \langle v_1, \ldots, v_n \rangle$ $(1 \leq i \leq n)$

$$(S, \texttt{let}\, \texttt{newrgn}\, \rho, x \,\texttt{in}\, e') \longmapsto_R (S\{\nu \mapsto \{\}\}, e'[\nu, \texttt{handle}(\nu)/\rho, x])$$
where $\nu \notin S$ and $\nu \notin e'$

$$(S, \texttt{let}\, \texttt{freergn}\, (\texttt{handle}(\nu)) \,\texttt{in}\, e') \longmapsto_R (S \backslash \nu, e')$$
where $\nu \in Dom(S)$

$$(S, \texttt{if}\ 0\ (e_2 \mid e_3)) \longmapsto_R (S, e_2)$$

$$(S, \texttt{if}\ i\ (e_2 \mid e_3)) \longmapsto_R (S, e_3)$$
where $i \neq 0$

$$(S, \nu.\ell[c_1, \ldots, c_m](v_1, \ldots, v_n)) \longmapsto_R (S, \theta_2(\theta_1(e)))$$
where $S(\nu.\ell) = \texttt{fix} f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
and $\theta_1 = [c_1, \ldots, c_m/\alpha_1, \ldots, \alpha_m]$ and $Dom(\Delta) = \alpha_1, \ldots, \alpha_m$
and $\theta_2 = [\nu.\ell, v_1, \ldots, v_n/f, x_1, \ldots, x_n]$

Figure 4.2: Capabilities: Operational Semantics

$\boxed{\Delta \vdash_R \Delta'}$

$$\frac{}{\Delta \vdash_R \cdot}\ (\texttt{R-ctxt-empty})$$

$$\frac{\Delta \vdash_R \Delta'}{\Delta \vdash_R \Delta', \alpha{:}\kappa}\ (\alpha \notin \mathit{Dom}(\Delta\Delta'))\ (\texttt{R-ctxt-var})$$

$$\frac{\Delta \vdash_R \Delta' \quad \Delta\Delta' \vdash_R C : \texttt{Store}}{\Delta \vdash_R \Delta', \epsilon \leq C}\ (\epsilon \notin \mathit{Dom}(\Delta\Delta'))\ (\texttt{R-ctxt-sub})$$

Figure 4.3: Capabilities: Static Semantics, Context Formation

that may be called when capability C is satisfied (see the next two subsections). The suffix "$\texttt{at}\, r$", like the corresponding suffix for tuple types, indicates the region in which the function is allocated.

As in the previous chapter, polymorphism plays a central role. Functions may be made polymorphic over types, regions or capabilities by adding a constructor context Δ to the function type as in $\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \texttt{at}\ r$. For convenience, types, regions and capabilities are combined into a single syntactic class of "constructors" and are distinguished by kinds. Thus, a type is a constructor with kind $\texttt{Type}$, a region is a constructor with kind $\texttt{Rgn}$, and a capability is a constructor with kind $\texttt{Store}$. We use the metavariable c to range over constructors, but use the metavariables τ, r and C when those constructors are types, regions and capabilities, respectively. We also use the metavariables ρ and ϵ for constructor variables of kind $\texttt{Rgn}$ and $\texttt{Store}$, and use the metavariable α for type variables and generic constructor variables.

For example, a polymorphic identity function that is allocated in region r, but whose continuation function may be in any region, may be given type

$$\forall[\alpha{:}\texttt{Type}, \rho{:}\texttt{Rgn}].(C, \alpha, (C, \alpha) \to \mathbf{0}\ \texttt{at}\ \rho) \to \mathbf{0}\ \texttt{at}\ r$$

for some appropriate C. Let f be such a function, let v be its argument with type τ, and let g be its continuation with type $(C, \tau) \to \mathbf{0}\ \texttt{at}\ r$. Then f is called by $f[\tau][r](v, g)$.

Figure 4.4 specifies all well-formed constructors and constructor contexts. The two main judgments $\Delta \vdash_R \Delta'$ and $\Delta \vdash_R c : \kappa$ assume that the constructor context Δ is well-formed. The first judgement states that Δ' is a well-formed constructor context and the second judgement states c is a well-formed constructor with kind κ. If Δ is a sequence of bindings of the form $\alpha_i{:}\kappa_i$ or $\alpha_i \leq C$ (where i ranges from 1 to n) then the domain of Δ is the sequence of constructor variables $\alpha_1, \ldots, \alpha_n$. Occasionally, we will use the notation $[c_1, \ldots, c_n/\Delta]$ to refer to the simultaneous capture-avoiding substitution $[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]$. We use the notation $\Delta\Delta'$ to indicate the constructor context formed by concatenating the elements of Δ' onto Δ. This notation is only defined if $\mathit{Dom}(\Delta) \cap \mathit{Dom}(\Delta') = \emptyset$.

4.2.3 Store Types

There are two different sorts of store types: one sort to give types to memory addresses and another sort to track the allocation status and aliasing of regions. Both sorts appear in the typing judgements for values and expressions so I will explain their structure here. For the

$\boxed{\Delta \vdash_R c : \kappa}$

$$\frac{}{\Delta \vdash_R \alpha : \kappa} \; (\Delta(\alpha) = \kappa) \; (\texttt{R-type-var})$$

$$\frac{}{\Delta \vdash_R \epsilon : \texttt{Store}} \; ((\epsilon \leq C) \in \Delta) \; (\texttt{R-type-sub})$$

$$\frac{}{\Delta \vdash_R \mathit{int} : \texttt{Type}} \; (\texttt{R-type-int})$$

$$\frac{\Delta \vdash_R r : \texttt{Rgn}}{\Delta \vdash_R \mathit{handle}(r) : \texttt{Type}} \; (\texttt{R-type-handle})$$

$$\frac{\Delta \vdash_R \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash_R r : \texttt{Rgn}}{\Delta \vdash_R \langle \tau_1, \ldots, \tau_n \rangle \; \texttt{at} \; r : \texttt{Type}} \; (\texttt{R-type-tuple})$$

$$\frac{\begin{array}{c}\Delta \vdash_R \Delta' \quad \Delta\Delta' \vdash_R \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \\ \Delta\Delta' \vdash_R C : \texttt{Store} \quad \Delta \vdash_R r : \texttt{Rgn}\end{array}}{\Delta \vdash_R \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \; \texttt{at} \; r : \texttt{Type}} \; (\texttt{R-type-arrow})$$

$$\frac{}{\Delta \vdash_R \nu : \texttt{Rgn}} \; (\texttt{R-type-name}) \qquad \frac{}{\Delta \vdash_R \emptyset : \texttt{Store}} \; (\texttt{R-type-}\emptyset)$$

$$\frac{\Delta \vdash_R r : \texttt{Rgn}}{\Delta \vdash_R \{r\} : \texttt{Store}} \; (\texttt{R-type-single})$$

$$\frac{\Delta \vdash_R C_1 : \texttt{Store} \quad \Delta \vdash_R C_2 : \texttt{Store}}{\Delta \vdash_R C_1 * C_2 : \texttt{Store}} \; (\texttt{R-type-plus})$$

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R \overline{C} : \texttt{Store}} \; (\texttt{R-type-bar})$$

Figure 4.4: Capabilities: Static Semantics, Type Formation

$\boxed{\vdash_R \Upsilon}$

$$\frac{\cdot \vdash_R \tau_i \qquad \text{(for } 1 \leq i \leq n\text{)}}{\vdash_R \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}} \ (\texttt{R-region-type})$$

$\boxed{\vdash_R \Psi}$

$$\frac{\vdash_R \Upsilon_i \qquad \text{(for } i \leq i \leq n\text{)}}{\vdash_R \{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}} \ (\texttt{R-memory-type})$$

Figure 4.5: Capabilities: Static Semantics, Memory Types

proof of soundness of the type system, we must also specify rules that relate the store to each of these two sorts of types. I will defer a formal discussion of the latter rules to section 4.2.6.

The first part requires considerably less complex type structure than the work of the previous chapter since I will not attempt to track aliasing between individual addresses here and the types of addresses will be *invariant* throughout their lifetimes. Type invariance obviates the need for dependency mechanisms and makes it possible to use simple store typing rules reminiscent of Harper's treatment of references in ML [Har94]. I define *memory types* as finite partial maps from region names to *region types* and region types as finite partial maps from locations to object types. The meta-variable Ψ ranges over memory types and Υ ranges over region types. Now, if a region named ν appears in the domain of the memory type Ψ and ℓ appears in the domain of region $\Psi(\nu)$, then the address $\nu.\ell$ has the type $\Psi(\nu.\ell)$, regardless of where the address appears in the program. The judgements for well-formedness of region and memory types appear in figure 4.5.

The second part of the store typing discipline involves tracking aliasing and allocation status of regions. Unlike address types, these properties *vary* from one program point to the next. To capture this variability, I use similar store typing mechanisms in this chapter (the C types) as in the previous one, on top of the memory types Ψ. To avoid confusion between the two cooperating mechanisms for store typing, both of which are slightly different from anything we have seen before, I will refer to the C types exclusively as capabilities and the Ψ types exclusively as memory types.

The structural rules for the capabilities of this chapter are somewhat more complex than in the previous chapter as there are both linear capabilities and a special form of non-linear (contractive) capability. I will defer discussion of the non-linear capabilities until a point when they can be properly motivated and explained (section 4.2.5). As for linear capabilities, they have the form $\emptyset$, $\{r\}$, ϵ or $C_1 * C_2$ where C_1 and C_2 are both capabilities. We can partially understand each of these capabilities in terms of the regions that they give access to. The empty capability, $\emptyset$, grants access to no regions. The singleton capability $\{r\}$ grants access to the single region r. As in the previous chapter, the capability ϵ is used to make functions store-polymorphic. It grants access to an unknown number of regions. Finally, $C_1 * C_2$ grants access to all the regions in C_1 or in C_2. Often, I abbreviate the capability $\{r_1\} * \cdots * \{r_n\}$ by $\{r_1, \ldots, r_n\}$.

I call these capabilities linear because, as for the store types of the previous chapter, the standard weakening and contraction rules are not admissible. Capabilities are, however, associative and commutative and therefore an exchange rule is admissible. The empty capability is the identity for the $*$ operator. Figure 4.6 defines the equality relation on linear capabilities.

$\boxed{\Delta \vdash_R c_1 = c_2 : \kappa}$

$$\frac{\Delta \vdash_R c : \kappa}{\Delta \vdash_R c = c : \kappa} \ (\texttt{R-eq-reflex})$$

$$\frac{\Delta \vdash_R c_2 = c_1 : \kappa}{\Delta \vdash_R c_1 = c_2 : \kappa} \ (\texttt{R-eq-symm})$$

$$\frac{\Delta \vdash_R c_1 = c_2 : \kappa \quad \Delta \vdash_R c_2 = c_3 : \kappa}{\Delta \vdash_R c_1 = c_3 : \kappa} \ (\texttt{R-eq-trans})$$

$$\frac{\Delta \vdash_R C_1 = C_1' : \texttt{Store} \quad \Delta \vdash_R C_2 = C_2' : \texttt{Store}}{\Delta \vdash_R C_1 * C_2 = C_1' * C_2' : \texttt{Store}} \ (\texttt{R-eq-congruence-join})$$

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R \emptyset * C = C : \texttt{Store}} \ (\texttt{R-eq-}\emptyset)$$

$$\frac{\Delta \vdash_R C_1 : \texttt{Store} \quad \Delta \vdash_R C_2 : \texttt{Store}}{\Delta \vdash_R C_1 * C_2 = C_2 * C_1 : \texttt{Store}} \ (\texttt{R-eq-comm})$$

$$\frac{\Delta \vdash_R C_i : \texttt{Store} \quad (\text{for } 1 \leq i \leq 3)}{\Delta \vdash_R (C_1 * C_2) * C_3 = C_1 * (C_2 * C_3) : \texttt{Store}} \ (\texttt{R-eq-assoc})$$

Figure 4.6: Capabilities: Static Semantics, Linear Capability Equality

4.2.4 Expression Typing

The central problem is how to ensure statically that no region is used after it is freed. The typing rules enforce this property using capabilities that specify the region accesses that are permitted. The main typing judgement is

$$\Psi; \Delta; \Gamma; C \vdash_R e$$

which states that (when the store has type Ψ, free constructor variables have kinds given by Δ and free value variables have types given by Γ) it is legal to execute the term e, *provided that the capability C is held.* A related typing judgement is

$$\Psi; \Delta; \Gamma; C \vdash_R d \Rightarrow \Delta'; \Gamma'; C'$$

which states that if the capability C is held, it is legal to execute the declaration d, which results in new constructor context Δ', new value context Γ' and new capability C'. Since Ψ is invariant, it does not appear on the right-hand side of this judgement.

In order to read a field from a tuple in region r, it is necessary to hold the capability to access r, as in the rule:

$$\frac{\begin{array}{c}\Delta \vdash_R C = C' * \{r\} : \texttt{Store} \\ \Psi; \Delta; \Gamma \vdash_R v : \langle \tau_1, \ldots, \tau_n \rangle \; \texttt{at} \; r\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R x = \pi_i(v) \Rightarrow \Delta; \Gamma\{x{:}\tau_i\}; C} \; (x \notin \mathit{Dom}(\Gamma) \wedge 1 \leq i \leq n)$$

The first subgoal indicates that the capability held (C) is equivalent to some capability that includes $\{r\}$.

A similar rule is used to allocate an object in a region. Since the type of a heap value reflects the region in which it is allocated, the heap value typing judgement (the second subgoal below) must be provided with that region.

$$\frac{\begin{array}{c}\Delta \vdash_R C = C' * \{r\} : \texttt{Store} \\ \Psi; \Delta; \Gamma \vdash_R h \; \texttt{at} \; r : \tau \\ \Psi; \Delta; \Gamma \vdash_R v : \mathit{handle}(r)\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R x = h \; \texttt{at} \; v \Rightarrow \Delta; \Gamma\{x{:}\tau\}; C} \; (x \notin \mathit{Dom}(\Gamma))$$

Functions Functions are defined by the following form

$$\texttt{fix} f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$$

where f stands for the function itself and may appear free in the body, Δ specifies the function's constructor arguments, and C is the function's capability precondition. When Δ is empty and f does not appear free in the function body I may abbreviate the `fix` form by $\lambda(C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$.

In order to call a function residing in region r, it is again necessary to hold the capability to access r, and also to hold a capability equivalent to the function's capability precondition:

$$\frac{\begin{array}{c}\Delta \vdash_R C = C'' * \{r\} : \texttt{Store} \quad \Delta \vdash_R C = C' : \texttt{Store} \\ \Psi; \Delta; \Gamma \vdash_R v : (C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \; \texttt{at} \; r \qquad \Psi; \Delta; \Gamma \vdash_R v_i : \tau_i\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R v(v_1, \ldots, v_n)}$$

The body of a function may then assume the function's capability precondition is satisfied, as indicated by the capability C in the premise of the rule:[5]

$$\frac{\Psi; \Delta; \Gamma\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C \vdash_R e}{\Psi; \Delta; \Gamma \vdash_R \lambda(C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \; \texttt{at} \; r : \tau_f} \; (x_i \notin Dom(\Gamma))$$

As might be expected, the annotation "$\texttt{at}\ r$" indicates that the closure value resides in region r. The resultant function type τ_f is $(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \; \texttt{at} \; r$.

As in previous chapters, we will extend the required capability for a function with a quantified capability variable. This variable may be instantiated with whatever capabilities are leftover after satisfying the required capability. Consequently, the function may be used in a variety of contexts. For example, functions with type

$$\forall[\epsilon{:}\texttt{Store}].(\{r\} * \epsilon, \ldots) \rightarrow \mathbf{0} \; \texttt{at} \; r$$

may be called with any capability that extends $\{r\}$.

When a function or continuation is polymorphic, its type constructor arguments may be instantiated one at a time, leading to partially-applied polymorphic functions with the form $v[c]$. The following rule gives a type to this sort of value:

$$\frac{\Psi; \Delta; \Gamma \vdash_R v : \forall[\alpha{:}\kappa, \Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \; \texttt{at} \; r \qquad \Delta \vdash c : \kappa}{\Psi; \Delta; \Gamma \vdash_R v[c] : (\forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0})[c/\alpha] \; \texttt{at} \; r}$$

The common case is still to apply multiple type arguments at once. We often abbreviate multiple type applications $v[c_1] \cdots [c_n]$ by $v[c_1, \ldots, c_n]$. As indicated in the rule for function call, a function must be fully-applied before it can be called.

Allocation and Deallocation The typing rule for region allocation is quite similar to the rule we saw for allocating new memory blocks. It extends the current capability with a new linear capability for the region that is allocated:

$$\frac{}{\begin{array}{l}\Psi; \Delta; \Gamma; C \vdash_R \texttt{newrgn} \; \rho, x \Rightarrow \\ \qquad\qquad \Delta, \rho{:}\texttt{Rgn}; \Gamma\{x{:}handle(\rho)\}; C * \{\rho\}\end{array}} \; (\rho \notin Dom(\Delta), x \notin Dom(\Gamma))$$

We already have a great deal of experience with deallocation so it is not surprising that the rule for region deallocation removes a linear capability for the region in question from the current context:

$$\frac{\Delta; \Gamma \vdash_R v : handle(r) \quad \Delta \vdash_R C = C' * \{r\} : \texttt{Store}}{\Psi; \Delta; \Gamma; C \vdash_R \texttt{freergn} \, v \Rightarrow \Delta; \Gamma; C'}$$

Intuitively, this rule is safe because the capability $\{r\}$ is linear and cannot appear elsewhere in C'. Since the capability does not appear in C', the following code does not have access to the region r.

[5] This rule specializes the full rule for *fix* to the case where the function is neither polymorphic nor recursive.

Shortcomings The type system I have defined to this point is sound and has some attractive garbage collection properties for regions. However, it is incomparable to Tofte and Talpin's type system. We need a type system that is at least as expression as Tofte and Talpin's language in order to use the region *inference* algorithms that have been developed by Tofte and others. To see where the current system falls short, consider the following perfectly reasonable increment function:

$$
\begin{array}{l}
\texttt{fix}\, incr\, [\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}](\{\rho_1, \rho_2\}, x{:}handle(\rho_1), y{:}\langle int\rangle\ \texttt{at}\ \rho_2, cont{:}\tau_{cont}). \\
\quad \texttt{let}\ z_1 = \pi_1 y\ \texttt{in} \\
\quad \texttt{let}\ z_2 = z_1 + 1\ \texttt{in} \\
\quad \texttt{let}\ z_3 = \langle z_2\rangle\ \texttt{at}\ x\ \texttt{in} \\
\quad cont(z_3)
\end{array}
$$

where $\tau_{cont} = (\{\rho_1, \rho_2\}, \langle int\rangle\ \texttt{at}\ \rho_1) \to void$.

This function is well-formed according to the typing rules: The function begins with the capability $\{\rho_1, \rho_2\}$, meaning that the regions ρ_1 and ρ_2 are both accessible. The function makes use of this fact when it extracts z from region ρ_2 and then allocates in region ρ_1 using the region handle x.

Now suppose we want to increment a value stored in some region r and place the result in the same region, rather than a different one. In other words, assume x' has type $handle(r)$ and y' has type $\langle int\rangle$ at r. Even if the current capability is $\{r\}$, we cannot call *incr*. The expression $incr[r, r](x', y', cont')$ for any continuation $cont'$ is ill-formed in a context where the current capability is $\{r\}$ since the capability $\{r\} * \{r\}$ is required by *incr* when ρ_1 and ρ_2 are both instantiated by r and we cannot use a contraction rule to relate these two capabilities. In other words,

$$\{r\} \neq \{r\} * \{r\}$$

After seeing this example, one's first thought might be that the linear discipline is unnecessarily strong. Perhaps, we should allow a contraction rule. Unfortunately, the solution is not so simple. If we were to allow contraction, we would quickly find ourselves in trouble with the following simple function:

$$
\begin{array}{l}
\texttt{fix}\, f\, [\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}](\{\rho_1, \rho_2\}, x{:}handle(\rho_1), y{:}\langle int\rangle\ \texttt{at}\ \rho_2, cont{:}\tau'_{cont}). \\
\quad \texttt{let freergn}\ x\ \texttt{in} \\
\quad \texttt{let}\ z_1 = \pi_1 y\ \texttt{in} \\
\quad cont(z_3)
\end{array}
$$

where $\tau'_{cont} = (\{\rho_2\}, \langle int\rangle\ \texttt{at}\ \rho_1) \to void$.

This function is well-formed according to the typing rules: The function begins with the capability $\{\rho_1, \rho_2\}$ and ρ_1 is removed by the `freergn` declaration, leaving $\{\rho_2\}$. The tuple y is allocated in ρ_2, so the projection is legal. However, if we allow a contraction rule of the form

$$\{r\} = \{r\} * \{r\}$$

then we can instantiate ρ_1 and ρ_2 with the same region r as in $f[r, r](x', y', cont')$. In other words, we can create a situation in which ρ_1 and ρ_2 alias one another. In this case, the `freergn` declaration will deallocate r and the projection will attempt to read from r, which is a run-time error.

$$\frac{\Delta \vdash_R C = C' : \texttt{Store}}{\Delta \vdash_R \overline{C} = \overline{C'} : \texttt{Store}} \ (\texttt{R-eq-congruence-bar})$$

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R \overline{C} = \overline{C} * \overline{C} : \texttt{Store}} \ (\texttt{R-eq-dup})$$

$$\frac{}{\Delta \vdash_R \overline{\emptyset} = \emptyset : \texttt{Store}} \ (\texttt{R-eq-bar-}\emptyset)$$

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R \overline{\overline{C}} = \overline{C} : \texttt{Store}} \ (\texttt{R-eq-bar-idem})$$

$$\frac{\Delta \vdash_R C_1 : \texttt{Store} \quad \Delta \vdash_R C_2 : \texttt{Store}}{\Delta \vdash_R \overline{C_1 * C_2} = \overline{C_1} * \overline{C_2} : \texttt{Store}} \ (\texttt{R-eq-distrib})$$

Figure 4.7: Capabilities: Static Semantics, Non-linear Capability Equality

This problem is a familiar one. To free a region safely it is necessary to delete all copies of the capability. If capabilities are linear, this can be handled easily. However, in order to type realistic code, we must have some way of dealing with region aliases, despite the fact that in the presence of polymorphic variables (such as ρ_1 and ρ_2), there may be no local analysis we can do to determine when two variables alias one another.

4.2.5 Non-linear Capabilities

My solution is to introduce a new form of non-linear capability, $\overline{C}$. Capabilities of this form resemble intuitionistic types ($!\tau$) in that they allow a special form of contraction, which I call the duplication rule:

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R \overline{C} = \overline{C} * \overline{C} : \texttt{Store}} \ (\texttt{R-eq-dup})$$

This rule makes it possible to create an unlimited number of aliases of a non-linear capability. As demonstrated by the function f above, such aliases are dangerous in the presence of deallocation. Hence, while non-linear capabilities grant the privilege to access objects within a region as well as the privilege to allocate within a region, they do not grant the privilege to deallocate a region. Only regions for which we have a linear capability may be deallocated.[6] The remaining equivalence rules for non-linear capabilities appear in Figure 4.7.

Suppose we rewrite the function *incr* so that it expects non-linear capabilities rather than linear ones:

[6] It is possible to deallocate regions with non-linear capabilities eventually. See section 4.2.5.

$$\texttt{fix}\, \mathit{incr}'\, [\rho_1{:}\mathbf{Rgn}, \rho_2{:}\mathbf{Rgn}](\overline{\{\rho_1, \rho_2\}}, x{:}\mathit{handle}(\rho_1), y{:}\langle \mathit{int} \rangle\ \texttt{at}\ \rho_2, \mathit{cont}{:}\tau_{\mathit{cont}}).$$

```
    let z1 = π1 y in
    let z2 = z1 + 1 in
    let z3 = ⟨z2⟩ at x in
    cont(z3)
```

where $\tau_{cont} = (\overline{\{\rho_1, \rho_2\}}, \langle int \rangle\ \texttt{at}\ \rho_1) \to \mathbf{0}$.

Intuitively, the function should continue to type check. In particular, the projection from region ρ_2 is safe and the allocation into ρ_1 is also safe since non-linear capabilities allow this sort of access. Furthermore, if we hold capability $\overline{\{r\}}$, we may call $incr'$ by instantiating ρ_1 and ρ_2 with r, since

$$\overline{\{r\}} = \overline{\{r\}} * \overline{\{r\}} = \overline{\{r\} * \{r\}}$$

On the other hand, we cannot rewrite the function f with non-linear capabilities because f deallocates one of its regions:

$$\texttt{fix}\, f'\, [\rho_1{:}\mathbf{Rgn}, \rho_2{:}\mathbf{Rgn}](\overline{\{\rho_1, \rho_2\}}, x{:}\mathit{handle}(\rho_1), y{:}\langle \mathit{int} \rangle\ \texttt{at}\ \rho_2, \mathit{cont}{:}\tau'_{\mathit{cont}}).$$

```
    let freergn x in      % Type check fails
    let z1 = π1 y in
    cont(z3)
```

Subcapabilities The capabilities $\{r\}$ and $\overline{\{r\}}$ are not the same, but the former should provide all the privileges of the latter. We therefore say that the former is a *subcapability* of the latter. We write subcapability judgements using the form $\Delta \vdash_R C \leq C'$. The principal rule relates any capability C to its non-linear relative $\overline{C}$:

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R C \leq \overline{C}}\ (\texttt{R-sub-bar})$$

In the complete type system, some of the access rules from Section 4.2.4 are modified to account for subcapabilities. These new rules allow us to check the $incr'$ function given above. For example, the allocation and projection rules become:

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_R v : \mathit{handle}(r) \\ \Psi; \Delta; \Gamma \vdash_R h\ \texttt{at}\ r : \tau \qquad \Delta \vdash C \leq C' * \overline{\{r\}}\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R x = h\ \texttt{at}\ v \Longrightarrow \Delta; \Gamma\{x{:}\tau\}; C}\ (x \notin \mathit{Dom}(\Gamma))\ (\texttt{R-hval})$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_R v : \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ r \\ \Delta \vdash C \leq C' * \overline{\{r\}}\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R x = \pi_i v \Longrightarrow \Delta; \Gamma\{x{:}\tau_i\}; C}\ (x \notin \mathit{Dom}(\Gamma) \wedge 1 \leq i \leq n)\ (\texttt{R-proj})$$

Both of these rules require that the current capability be less than some capability that includes the region r that is being accessed.

In contrast to these new rules, the rule for deallocation is unchanged in the presence of subcapabilities. It continues to require that the deallocated region be linear. Therefore, the

unsafe function f' will not type check. We are saved because there is no capability C' (in particular $C' = \overline{\{\rho_2\}}$ is insufficient) such that

$$\overline{\{\rho_1, \rho_2\}} = C' * \{\rho_1\}$$

The rule for function calls must also be modified slightly to take advantage of subcapabilities. Like a projection, a function call reads from a region and this region may be non-linear. Hence the complete type system uses the following rule:

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_R v : (C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ r \\ \Psi; \Delta; \Gamma \vdash_R v_i : \tau_i \quad (\text{for } 1 \leq i \leq n) \\ \Delta \vdash C \leq C'' * \overline{\{r\}} \qquad \Delta \vdash C = C' : \mathtt{Store}\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R v(v_1, \ldots, v_n)} \ (\mathtt{R\text{-}app})$$

This rule relates the current capability C to the required capability C' via an equality judgement. Therefore, if we hold a non-linear capability, we can make as many copies of it as we need to satisfy the required capability on a function like *incr'*. However, what happens if we hold a linear capability rather than a non-linear one? Can we use the function *incr'*? The immediate answer is no: $\{r\} \neq \overline{\{r, r\}}$.

Perhaps we should relax this rule by replacing the equality with an inequality judgement. If we choose this path, upon a function call, linear capabilities could be replaced by non-linear capabilities. For example, if the current capability was $\{r\}$ then it would be possible to call a function with the following type

$$\forall[\rho_1{:}\mathbf{Rgn}, \rho_2{:}\mathbf{Rgn}].(\overline{\{\rho_1, \rho_2\}}, \ldots) \rightarrow \mathbf{0}\ \mathtt{at}\ r$$

by instantiating ρ_1 and ρ_2 with r and using the inequality relation:

$$\{r\} \leq \overline{\{r\}} = \overline{\{r\}} * \overline{\{r\}} = \overline{\{r\} * \{r\}}$$

Unfortunately, this solution, while sound, simply raises other difficulties. It transforms linear regions that can be deallocated into non-linear regions that can *never* be deallocated and so it is almost always a mistake. For example, suppose we hold the capability $\{r\}$ and a function g has type:

$$\forall[\rho_1{:}\mathbf{Rgn}, \rho_2{:}\mathbf{Rgn}].(\overline{\{\rho_1, \rho_2\}}, \ldots, (\overline{\{\rho_1, \rho_2\}}, \ldots) \rightarrow \mathbf{0}\ \mathtt{at}\ \rho_1) \rightarrow \mathbf{0}\ \mathtt{at}\ r$$

We could use the subcapability relation to call this function by instantiating ρ_1 and ρ_2 with r. However, the continuation would be unable to free r when the region was logically dead. The continuation only possesses the non-linear capability $\overline{\{r, r\}} = \overline{\{r\}}$, not the linear capability $\{r\}$ necessary to free the region. Moreover, it does not help to strengthen the capability of the continuation to (for example) $\{r\}$, because then g may not call it (g itself only possessing the capability $\overline{\{r, r\}}$).

We *may* recover uniqueness information by quantifying a capability variable. Suppose we again hold capability $\{r\}$ and the function g' has type:

$$\forall[\rho_1{:}\mathbf{Rgn}, \rho_2{:}\mathbf{Rgn}, \epsilon{:}\mathtt{Store}].(\epsilon, \ldots, (\epsilon, \ldots) \rightarrow \mathbf{0}\ \mathtt{at}\ \rho_1) \rightarrow \mathbf{0}\ \mathtt{at}\ r$$

We may instantiate ϵ with $\{r\}$ and then the continuation will possess that same capability, allowing it to free r. Unfortunately, the body of function g' no longer has the capability to access ρ_1 and ρ_2, since its type draws no connection between them and ϵ.

$\boxed{\Delta \vdash C_1 \leq C_2}$

$$\frac{\Delta \vdash_R C_1 = C_2 : \texttt{Store}}{\Delta \vdash_R C_1 \leq C_2} \ (\texttt{R-sub-eq})$$

$$\frac{\Delta \vdash_R C_1 \leq C_2 \quad \Delta \vdash_R C_2 \leq C_3}{\Delta \vdash_R C_1 \leq C_3} \ (\texttt{R-sub-trans})$$

$$\frac{\Delta \vdash_R C_1 \leq C_1' \quad \Delta \vdash_R C_2 \leq C_2'}{\Delta \vdash_R C_1 * C_2 \leq C_1' * C_2'} \ (\texttt{R-sub-congruence-join})$$

$$\frac{\Delta \vdash_R C \leq C'}{\Delta \vdash_R \overline{C} \leq \overline{C'}} \ (\texttt{R-sub-congruence-bar})$$

$$\frac{}{\Delta \vdash_R \epsilon \leq C} \ ((\epsilon \leq C) \in \Delta) \ (\texttt{R-sub-var})$$

$$\frac{\Delta \vdash_R C : \texttt{Store}}{\Delta \vdash_R C \leq \overline{C}} \ (\texttt{R-sub-bar})$$

Figure 4.8: Capabilities: Static Semantics, Subcapability Relation

Bounded Quantification We solve these problems by using bounded quantification to relate ρ_1, ρ_2 and ϵ. Suppose h has type:

$$\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}, \epsilon \leq \overline{\{\rho_1, \rho_2\}}].(\epsilon, \ldots, (\epsilon, \ldots) \to \mathbf{0} \texttt{ at } \rho_1) \to \mathbf{0} \texttt{ at } r$$

If we hold capability $\{r\}$, we may call h by instantiating ρ_1 and ρ_2 with r and instantiating ϵ with $\{r\}$. This instantiation is permissible because $\{r\} \leq \overline{\{r, r\}}$. As with g, the continuation will possess the capability $\{r\}$, allowing it to free r, but the body of h (like that of f) will have the capability to access ρ_1 and ρ_2, since $\epsilon \leq \overline{\{\rho_1, \rho_2\}}$. The complete subcapability relation, including a rule for bounded quantification, is defined formally in Figure 4.8.

Bounded quantification solves the problem by revealing some information about a capability ϵ, while still requiring the function to be parametric over ϵ. Hence, when the function calls its continuation we regain the stronger capability (to free r), although that capability was temporarily hidden in order to duplicate r. More generally, bounded quantification allows us to hide some privileges when calling a function, and regain those privileges in its continuation. Thus, we support statically checkable attenuation and amplification of capabilities.

Static Semantics So Far Together, bounded parametric polymorphism, and notions of linearity and aliasing provide a flexible language for expressing the lifetimes of regions. Figures 4.9, 4.10, 4.11 formally summarize the rules for type checking instructions and values that depend upon these concepts. We have already explained the majority of these rules in previous sections and the rules that we have not yet specified are the obvious ones (integers are given type *int*, etc.). Notice, however, that the form of the judgement for heap values h is slightly different from the judgements for instructions and small values v. The judgment $\Psi; \Delta; \Gamma \vdash_R h \texttt{ at } r : \tau$

states that when memory has type Ψ, free constructor variables have kinds given by Δ and free value variables have types given by Γ, the heap value h resides in region r and has type τ.

4.2.6 Run-time Values and Store Typing

The previous section contains all of the information programmers or compilers require to write type-safe programs in the Capability Calculus. However, in order to prove a type soundness result in the style of Wright and Felleisen [WF94], we must be able to type check programs at every step during their evaluation. In this section, we give the static semantics of the store and of the run-time values (including concrete region handles and addresses) that are not normally manipulated by programmers, but are nevertheless necessary to prove our soundness result.

Figure 4.12 specifies the rules for typing memory, most of which are straightforward. The judgment $\vdash_R S : \Psi$ states that S is described by Ψ and the judgement $\Psi \vdash_R R\,\texttt{at}\,\nu : \Upsilon$ states that region R with name ν is described by Υ. Informally, these judgements ensure that for addresses $\nu.\ell$, $\Psi(\nu.\ell)$ is type τ if and only if the store S described by Ψ contains a value v at address $\nu.\ell$ that has type τ.

The next judgment, $\Psi \vdash_R C$ sat, is called the satisfiability judgment and it formalizes the connection between the static capability and the run-time state of memory. Clearly, the current capability must not contain any regions that are not in the store; this could lead to a runtime error. However, it is equally important that the store not contain regions for which we have no capability as such regions can never be freed. Consequently, satisfiability ensures that at any time during execution of a program, our capability is equal to $\{r_1, \ldots, r_n\}$ where each r_i occurs exactly once in the current memory. Furthermore, by virtue of the fact that $\cdot \vdash_R \{r\} \neq \{r\} * \{r\} : \texttt{Store}$, no region may appear more than once in C. Each of these properties are essential to ensure that regions are used safely.

Figure 4.13 contains rules for small values that only appear at run time (addresses and region handles). The rules for typing an address $\nu.\ell$ are quite unusual, but crucial to the type soundness proof. The first rule, v-addr, is used during the lifetime of the region ν: If the region ν is in memory then ν will also be in the domain of the memory type Ψ. Therefore, rule v-addr applies and $\nu.\ell$ will have type $\Psi(\nu.\ell)$. Now consider some point in the computation after the region ν has been deallocated. The region ν is no longer in the memory, but the addresses $\nu.\ell$ may still appear embedded in tuples or closures allocated in other regions, and, therefore, they must be given types. If a region ν does not appear in memory type Ψ, the type system has the flexibility to give $\nu.\ell$ *any* function type (by rule $\texttt{R} - \texttt{v} - \texttt{addr} - \texttt{arrow}$) or tuple type (by rule $\texttt{R} - \texttt{v} - \texttt{addr} - \texttt{tuple}$).

At first glance, these rules would appear to lead to unsoundness: The address $\nu.\ell$ is a dangling pointer and it may be given a valid type. Fortunately, though, capabilities prevent anything from going wrong. The satisfiability judgment ensures that programs only ever possess capabilities for regions that appear in the store, and, as we explained earlier, programs can only access the regions for which they have capabilities. Consequently, a dangling pointer may be given a valid tuple or function type, but capabilities prevent it from being accessed.

We now have all components necessary to define a well-formed program. The program (S, e) is well-formed if S can be described by a well-formed heap type Ψ, there exists a capability C such that C *satisfies* the heap type Ψ, and finally, the expression e is well-formed with respect to Ψ and C:

$$\frac{\vdash_R S : \Psi \qquad \Psi \vdash_R C \text{ sat} \qquad \Psi; \cdot; \cdot; C \vdash_R e}{\vdash_R (S, e)} \; (\texttt{R-program})$$

$\boxed{\Psi;\Delta;\Gamma \vdash_R h \;\texttt{at}\; r : \tau}$

$$\frac{\begin{array}{c}\Delta \vdash_R \tau_f : \texttt{Type} \\ \Psi;\Delta\Delta';\Gamma\{f{:}\tau_f, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\};C \vdash_R e \\ \begin{pmatrix}\tau_f = \forall[\Delta'].(C,\tau_1,\ldots,\tau_n) \rightarrow \mathbf{0} \;\texttt{at}\; r \\ f, x_1, \ldots, x_n \notin Dom(\Gamma)\end{pmatrix}\end{array}}{\Psi;\Delta;\Gamma \vdash_R \texttt{fix} f[\Delta'](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \;\texttt{at}\; r : \tau_f} \;(\texttt{R-h-fix})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v_i : \tau_i \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash_R r : \texttt{Rgn}}{\Psi;\Delta;\Gamma \vdash_R \langle v_1, \ldots, v_n\rangle \;\texttt{at}\; r : \langle \tau_1, \ldots, \tau_n\rangle \;\texttt{at}\; r} \;(\texttt{R-h-tuple})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R h \;\texttt{at}\; r : \tau' \qquad \Delta \vdash_R \tau' = \tau : \texttt{Type}}{\Psi;\Delta;\Gamma \vdash_R h \;\texttt{at}\; r : \tau} \;(\texttt{R-h-eq})$$

$\boxed{\Psi;\Delta;\Gamma \vdash_R v : \tau}$

$$\frac{}{\Psi;\Delta;\Gamma \vdash_R x : \tau} \;(\Gamma(x) = \tau) \;(\texttt{R-v-var})$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash_R i : int} \;(\texttt{R-v-int})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : \forall[\alpha{:}\kappa, \Delta'].(C,\tau_1,\ldots,\tau_n) \rightarrow \mathbf{0} \;\texttt{at}\; r \qquad \Delta \vdash c : \kappa}{\Psi;\Delta;\Gamma \vdash_R v[c] : (\forall[\Delta'].(C,\tau_1,\ldots,\tau_n) \rightarrow \mathbf{0})[c/\alpha] \;\texttt{at}\; r} \;(\texttt{R-v-type})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : \forall[\epsilon \leq C'', \Delta'].(C',\tau_1,\ldots,\tau_n) \rightarrow \mathbf{0} \;\texttt{at}\; r \qquad \Delta \vdash C \leq C''}{\Psi;\Delta;\Gamma \vdash_R v[C] : (\forall[\Delta'].(C',\tau_1,\ldots,\tau_n) \rightarrow \mathbf{0})[C/\epsilon] \;\texttt{at}\; r} \;(\texttt{R-v-sub})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : \tau' \quad \Delta \vdash_R \tau' = \tau : \texttt{Type}}{\Psi;\Delta;\Gamma \vdash_R v : \tau} \;(\texttt{R-v-eq})$$

Figure 4.9: Capabilities: Static Semantics, Heap and Word Values

$$\boxed{\Psi;\Delta;\Gamma;C \vdash_R d \Longrightarrow \Delta';\Gamma';C'}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : \tau}{\Psi;\Delta;\Gamma;C \vdash_R x = v \Longrightarrow \Delta;\Gamma\{x{:}\tau\};C}\ (x \notin Dom(\Gamma))\ (\texttt{R-val})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v_1 : int \quad \Psi;\Delta;\Gamma \vdash_R v_2 : int}{\Psi;\Delta;\Gamma;C \vdash_R x = v_1\ p\ v_2 \Longrightarrow \Delta;\Gamma\{x{:}int\};C}\ (x \notin Dom(\Gamma))\ (\texttt{R-prim})$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash_R v : handle(r) \\ \Psi;\Delta;\Gamma \vdash_R h\ \texttt{at}\ r : \tau \qquad \Delta \vdash C \leq C' * \overline{\{r\}}\end{array}}{\Psi;\Delta;\Gamma;C \vdash_R x = h\ \texttt{at}\ v \Longrightarrow \Delta;\Gamma\{x{:}\tau\};C}\ (x \notin Dom(\Gamma))\ (\texttt{R-hval})$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash_R v : \langle\tau_1,\ldots,\tau_n\rangle\ \texttt{at}\ r \\ \Delta \vdash C \leq C' * \overline{\{r\}}\end{array}}{\Psi;\Delta;\Gamma;C \vdash_R x = \pi_i v \Longrightarrow \Delta;\Gamma\{x{:}\tau_i\};C}\ \begin{pmatrix} x \notin Dom(\Gamma) \\ 1 \leq i \leq n \end{pmatrix}\ (\texttt{R-proj})$$

$$\frac{}{\begin{array}{l}\Psi;\Delta;\Gamma;C \vdash_R \texttt{newrgn}\ \rho, x \Longrightarrow \\ \qquad \Delta\{\rho{:}\texttt{Rgn}\};\Gamma\{x{:}handle(\rho)\};C * \{\rho\}\end{array}}\ \begin{pmatrix} \rho \notin Dom(\Delta) \\ x \notin Dom(\Gamma) \end{pmatrix}\ (\texttt{R-newrgn})$$

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : handle(r) \quad \Delta \vdash C = C' * \{r\} : \texttt{Store}}{\Psi;\Delta;\Gamma;C \vdash_R \texttt{freergn}\, v \Longrightarrow \Delta;\Gamma;C'}\ (\texttt{R-freergn})$$

Figure 4.10: Capabilities: Static Semantics, Declarations

$\boxed{\Psi; \Delta; \Gamma; C \vdash_R e}$

$$\frac{\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C' \qquad \Psi; \Delta'; \Gamma'; C' \vdash_R e}{\Psi; \Delta; \Gamma; C \vdash_R \texttt{let}\, d\, \texttt{in}\, e} \; (\texttt{R-letdec})$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_R v : int \\ \Psi; \Delta; \Gamma; C \vdash_R e_2 \qquad \Psi; \Delta; \Gamma; C \vdash_R e_3\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R \texttt{if}\ v\ (e_2 \mid e_3)} \; (\texttt{R-if})$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_R v : (C', \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \texttt{at}\ r \\ \Psi; \Delta; \Gamma \vdash_R v_i : \tau_i \quad (\text{for } 1 \le i \le n) \\ \Delta \vdash C \le C'' * \overline{\{r\}} \qquad \Delta \vdash C = C' : \texttt{Store}\end{array}}{\Psi; \Delta; \Gamma; C \vdash_R v(v_1, \ldots, v_n)} \; (\texttt{R-app})$$

$$\frac{\Psi; \Delta; \Gamma \vdash_R v : int \quad \Delta \vdash C = \emptyset : \texttt{Store}}{\Psi; \Delta; \Gamma; C \vdash_R \texttt{halt} v} \; (\texttt{R-halt})$$

Figure 4.11: Capabilities: Static Semantics, Expressions

$\boxed{\Psi \vdash_R R\, \texttt{at}\, \nu : \Upsilon}$

$$\frac{\Psi; \cdot; \cdot \vdash_R h_i\ \texttt{at}\ \nu : \tau_i \quad (\text{for } 1 \le i \le n)}{\Psi \vdash_R \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}\ \texttt{at}\ \nu : \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}} \; (\texttt{R-region})$$

$\boxed{\vdash_R M : \Psi}$

$$\frac{\begin{array}{c}\vdash_R \Psi \\ \Psi \vdash_R R_i\ \texttt{at}\ \nu_i : \Upsilon_i \quad (\text{for } 1 \le i \le n)\end{array}}{\vdash_R \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \Psi} \; (\Psi = \{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}) \; (\texttt{R-memory})$$

$\boxed{\Psi \vdash_R C \text{ sat}}$

$$\frac{\cdot \vdash_R C = \{\nu_1, \ldots, \nu_n\} : \texttt{Store}}{\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash_R C \text{ sat}} \; (\texttt{R-sat})$$

Figure 4.12: Capabilities: Static Semantics, Memory

$\boxed{\Psi; \Delta; \Gamma \vdash_R v : \tau}$

$$\frac{}{\Psi; \Delta; \Gamma \vdash_R \nu.\ell : \tau} \ (\Psi(\nu.\ell) = \tau) \ (\texttt{R-v-addr})$$

$$\frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle \ \texttt{at} \ \nu : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash_R \nu.\ell : \langle \tau_1, \ldots, \tau_n \rangle \ \texttt{at} \ \nu} \ (\nu \notin Dom(\Psi)) \ (\texttt{R-v-addr-tuple})$$

$$\frac{\Delta \vdash \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \ \texttt{at} \ \nu : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash_R \nu.\ell : \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \ \texttt{at} \ \nu} \ (\nu \notin Dom(\Psi)) \ (\texttt{R-v-addr-arrow})$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash_R \texttt{handle}(\nu) : \nu \ \texttt{handle}} \ (\texttt{R-v-handle})$$

Figure 4.13: Capabilities: Static Semantics, Run-time Values

4.3 Properties of the Capability Calculus

As in previous chapters, I will focus on two properties of the Capability Calculus: Type Soundness and Complete Collection.

4.3.1 Type Soundness

As before, Type Soundness states that a program will never enter a *stuck state* during execution. A state (S, e) is *stuck* if there does not exist (S', e') such that $(S, e) \longmapsto (S', e')$ and e is not `halt` i. For example, a state that tries to project a value from a tuple that does not appear in memory is stuck.

Theorem 17 (Type Soundness)
If $\vdash_R \texttt{P}$ *and* $\texttt{P} \longmapsto_R^* \texttt{P}'$ *then* $\texttt{P}'$ *is not stuck.*

In the previous sections of this chapter, we have explained how to type memory, how to relate the memory typing to static capabilities and finally, given a collection of capabilities, how the rules for typing expressions prevent unsafe accesses to the store. These invariants are the main elements in the formal proof of soundness. However, there are many details to fill in. The proof is in the style of Wright and Felleisen [WF94] and uses the standard Type Preservation and Progress lemmas. Progress states that well-typed states are not stuck, and Preservation states that evaluation steps preserve well-typedness.

Lemma 18 (Type Preservation)
If $\vdash_R \texttt{P}$ *and* $\texttt{P} \longmapsto_R \texttt{P}'$ *then* $\vdash_R \texttt{P}'$

Lemma 19 (Progress) *If* $\vdash_R (S, e)$ *then either:*

1. *There exists* $\texttt{P}'$ *such that* $(S, e) \longmapsto_R \texttt{P}'$*, or*

2. $e = \texttt{halt}\ i$

The proofs of these lemmas and the soundness theorem itself appear in Appendix B.

4.3.2 Complete Collection

A second important property of the language is that well-typed terminating programs return all of their memory resources to the system before they halt.

Theorem 20 (Complete Collection) *If* $\vdash_R \texttt{P}$ *then either* P *diverges or* $\texttt{P} \longmapsto_R^* (\{\,\}, \texttt{halt}\ i)$.

By Subject Reduction and Progress, terminating programs end in well-formed machine states $(S, \texttt{halt}\ i)$. The typing rule for the `halt` expression requires that the capability C be empty. Using this fact, we can infer that the store S contains no regions. Appendix B also contains a formal proof of this theorem.

4.4 Examples

Example 1 Figure 4.14 shows an example program, including a function `count` that counts down to zero. The program begins by allocating regions ρ_1 and ρ_2 using the **newrgn** declaration, and puts the closure for `count` into ρ_1. The `count` function takes two arguments, a handle for region ρ and an integer reference x allocated in region ρ. If x is nonzero, `count` decrements it, storing the result again in ρ, and recurses. The `count` function requires a capability ϵ' at least as good as the capability $\{\rho_1, \rho, \rho_{cont}\}$ needed to access itself, its argument, and its continuation; and it passes on that same capability ϵ' to its continuation k. As we type check the body of the count function, we verify that we possess the necessary capabilities. Comments in the code indicate where these checks occur.

After defining the `count` function, we allocate another region (ρ_3) that will hold the continuation closure (cont). This continuation requires the capability $\{\rho_1, \rho_2, \rho_3\}$ in order to free the three regions. The last line of the code in figure 4.14 is a function application. At this point, `count` is passed the primary argument `ten`, the continuation "count" and a handle for the region ρ_2. We instantiate `count`'s capability, ϵ', with the current capability $\{\rho_1, \rho_2, \rho_3\}$, which a subcapability of the required capability $\emptyset * \{\rho_1, \rho_2, \rho_3\}$.

Example 2 In the first example, the `count` function uses all of the regions that are currently allocated and the capability variable ϵ is redundant. When the code instantiates ϵ at the call site for `count`, it does so with exactly the regions ρ_1, ρ, and ρ_{cont} which already appear in the bound on ϵ'. However, in general, ϵ will hide some "left-over" capability. For example, if we had allocated a fourth region, ρ_4, we would need to instantiate ϵ with the capability $\{\rho_4\}$ and make corresponding changes to the continuation. Now, ϵ would hide the capability on the fourth region from `count` but preserve it across the call so it could be deallocated in the continuation:

```
%%% count with ε hiding a left-over capability
let newrgn ρ1, xρ1 in
let newrgn ρ2, xρ2 in
let newrgn ρ3, xρ3 in
let newrgn ρ4, xρ4 in
let count = ... as before ...
% capability held is {ρ1, ρ2, ρ3, ρ4}
let ten = ⟨10⟩ at xρ2 in
let cont =
  (λ ({ρ1, ρ2, ρ3, ρ4}) ...)  at xρ2
```

```
let newrgn $\rho_1, x_{\rho_1}$ in
let newrgn $\rho_2, x_{\rho_2}$ in
% capability held is $\{\rho_1, \rho_2\}$
let count =
  (fix count
       [$\rho$:Rgn, $\rho_{cont}$:Rgn, $\epsilon$:Store, $\epsilon' \leq \overline{\epsilon * \{\rho_1, \rho, \rho_{cont}\}}$]
       ($\epsilon'$, $x_\rho$:$handle(\rho)$, $x$:$\langle int \rangle$ at $\rho$, $k$:$(\epsilon') \rightarrow \mathbf{0}$ at $\rho_{cont}$) .
    % capability held is $\epsilon' \leq \overline{\epsilon * \{\rho_1, \rho, \rho_{cont}\}}$
    let $n$ = $\pi_1(x)$ in                                          % $\rho$ ok
      if0 $n$
        ( $k()$                                                      % $\rho_{cont}$ ok
        |
          let $n'$ = $n - 1$ in
          let $x'$ = $\langle n' \rangle$ at $x_\rho$ in                              % $\rho$ ok
            count [$\rho, \rho_{cont}, \emptyset, \epsilon'$] ($x_\rho, x', k$))              % $\rho_1$ ok
  ) at $x_{\rho_1}$ in
let newrgn $\rho_3, x_{\rho_3}$ in
% capability held is $\{\rho_1, \rho_2, \rho_3\}$
let ten = $\langle 10 \rangle$ at $x_{\rho_2}$ in
let cont =
  ($\lambda$ ($\{\rho_1, \rho_2, \rho_3\}$) .
    % capability held is $\{\rho_1, \rho_2, \rho_3\}$
    let freergn $x_{\rho_3}$ in                                      % $\rho_3$ unique
    let freergn $x_{\rho_2}$ in                                      % $\rho_2$ unique
    let freergn $x_{\rho_1}$ in                                      % $\rho_1$ unique
      halt 0
  ) at $x_{\rho_3}$
in
  count [$\rho_2, \rho_3, \emptyset, \{\rho_1, \rho_2, \rho_3\}$] ($x_{\rho_2}$, ten, cont)
```

Figure 4.14: The Function count

```
in
  count [ρ2, ρ3, {ρ4}, {ρ4} * {ρ1, ρ2, ρ3}] (x_ρ2, ten, cont)
```

Example 3 The power of bounded quantification comes into play when a function is called with several regions, some of which may or may not be the same. For example, the original code could be rewritten to have `ten` and `cont` share a region, without changing the function `count` in any way:

```
%%% count with ten and cont sharing ρ2
let newrgn ρ1, x_ρ1 in
let newrgn ρ2, x_ρ2 in
let count = ... as before ...
% capability held is {ρ1, ρ2}
let ten = ⟨10⟩ at x_ρ2 in
let cont =
  (λ ({ρ1, ρ2}) ...)  at x_ρ2
in
  count [ρ2, ρ2, ∅, {ρ1, ρ2}] (x_ρ2, ten, cont)
```

In this example, ρ_{cont} is instantiated with ρ_2 and ϵ' is instantiated with $\{\rho_1, \rho_2\}$ (which is again the capability required by `cont`). However, `count` proceeds exactly as before because ϵ' is still as good as $\overline{\{\rho_1, \rho, \rho_{cont}\}}$ since:

$$
\begin{array}{rcl}
\{\rho_1, \rho_2\} & \leq & \overline{\{\rho_1, \rho_2\}} \\
& = & \overline{\{\rho_1, \rho_2, \rho_2\}} \\
& = & \overline{\emptyset} * \overline{\{\rho_1, \rho_2, \rho_2\}} \\
& = & \overline{\emptyset * \{\rho_1, \rho_2, \rho_2\}} \\
& = & (\overline{\epsilon * \{\rho_1, \rho_2, \rho_2\}})[\emptyset/\epsilon]
\end{array}
$$

Example 4 In the examples above, even though `count` is tail-recursive, we allocate a new cell each time around the loop and we do not deallocate any of the cells until the count is complete. However, since ρ never contains any live values other than the current argument, it is safe to reduce the program's space usage by deallocating the argument's region each time around the loop, as shown in Figure 4.15. Note that this optimization is not possible when region lifetimes must be lexically scoped.

In order to deallocate its argument, the revised `count` requires a unique capability for its argument's region ρ. Note that if the program were again rewritten so that `ten` and `cont` shared a region (which would lead to a run-time error, since `ten` is deallocated early), the program would no longer typecheck, since $\{\rho_1, \rho_2\} \not\leq \overline{\{\rho_1, \rho_2\}} * \{\rho_2\}$. However, the program rewritten so that `count` and `cont` share a region does not fail at run time, and does typecheck, since $\{\rho_1, \rho_2\} \leq \overline{\{\rho_1, \rho_1\}} * \{\rho_2\}$.

4.5 Discussion

There are several ways to extend the capability system developed in this chapter and a number of directions for future research. This section discusses some of these extensions and research possibilities as well as related work.

let newrgn ρ_1, x_{ρ_1} in
let newrgn ρ_2, x_{ρ_2} in
% capability held is $\{\rho_1, \rho_2, \rho_3\}$
let count =
(fix count
$[\rho{:}\mathbf{Rgn}, \rho_{cont}{:}\mathbf{Rgn}, \epsilon \leq \overline{\{\rho_1, \rho_{cont}\}}]$
$(\epsilon * \{\rho\}, x_\rho{:}handle(\rho), x{:}\langle int \rangle$ at $\rho,$
$k{:}(\epsilon) \rightarrow \mathbf{0}$ at $\rho_{cont})$.
% capability held is $\epsilon * \{\rho\}$
let n = $\pi_1(x)$ in % ρ ok
let freergn x_ρ in % ρ unique
% capability held is ϵ
if0 n
then $k()$ % ρ_{cont} ok
else
let n' = $n - 1$ in
let newrgn $\rho', x_{\rho'}$ in
% capability held is $\epsilon * \{\rho'\}$
let x' = $\langle n' \rangle$ at $x_{\rho'}$ in % ρ' ok
count $[\rho', \rho_{cont}, \epsilon]$ $(x_{\rho'}, x', k)$ % ρ_1 ok
) at x_{ρ_1} in
let ten = $\langle 10 \rangle$ at x_{ρ_2} in
let newrgn ρ_3, x_{ρ_3} in
let cont =
(λ $(\{\rho_1, \rho_3\})$.
% capability held is $\{\rho_1, \rho_3\}$
let freergn x_{ρ_3} in % ρ_3 unique
let freergn x_{ρ_1} in % ρ_1 unique
halt 0
) at x_{ρ_3}
in
count $[\rho_2, \rho_3, \{\rho_1, \rho_3\}]$ $(x_{\rho_2}$, ten, cont)

Figure 4.15: Count with Efficient Memory Usage

4.5.1 Aggregate Data Structures

The region inference algorithm used in the ML Kit with Regions (or simply "the Kit") [TBE+98] all the nodes in an aggregate data structure in the same region. Intuitively, their algorithm might give β-lists the following recursive type:

$$List[r, \beta] = \mu\alpha.\langle \mathcal{S}(1)\rangle \texttt{ at } r \cup \langle \mathcal{S}(2), \beta, \alpha\rangle \texttt{ at } r$$

This type indicates that all of the cons cells in the list inhabit the same region (the region r). When one aggregate data structure contains another aggregate data structure, the Kit may place them in different regions. Hence, a list of lists could have the following form:

$$\begin{array}{rcl} Listof\,List[r_1, r_2, \beta] & = & List[r_1, List[r_2, \beta]] \\ & = & \mu\alpha.\langle \mathcal{S}(1)\rangle \texttt{ at } r_1 \cup \langle \mathcal{S}(2), List[r_2, \beta], \alpha\rangle \texttt{ at } r_1 \end{array}$$

This organization is very effective when all of the nodes in a list, tree or other aggregate have the same or similar lifetimes. However, a different strategy is required when the lifetimes of nodes in the aggregate differ. This may well be the case in a long-lived data structure that has insert, lookup and delete functions. From time to time, nodes are inserted into the data structure and then later removed. However, if we use the type structure explained above then when a node is removed, it cannot be deallocated because it continues to inhabit the same region as all the other nodes in the data structure. In fact, no nodes can be deleted until the entire long-lived data structure is disposed of.

Tofte and others [TBE+98] have developed some programming techniques that often help avoid this sort of pitfall in practice. One such technique is to mimic a copying garbage collector. Once in a while, the programmer explicitly copies the entire aggregate data structure from the old region (analogous to the copying collector's from space) to a freshly allocated region (analogous to the copying collector's to space) and then deallocates the old region. One disadvantage of using this clever trick is that the programmer must track down all references into the old region themselves and make sure these references are never used in the future. Otherwise, the old region will not be able to be deallocated. A second disadvantage is that if the aggregate is a very large data structure then copying it will be expensive. As is the case for any copying garbage collector, space proportional to twice the size of the structure is required.

An alternative solution is to combine my region framework with the alias types described in chapter 3. Alias types allow fine-grained reuse that complements the courser-grained region approach. The key is to use a (linear) existential type to encapsulate the region used to store each node of the data structure. For example, here is an alternative list definition:

$$\begin{array}{l} List'[\beta] = \\ \quad \mu\alpha.\exists[\rho \mid \{\rho\}].\langle \mathcal{S}(1), \rho \texttt{ handle}\rangle \texttt{ at } \rho \cup \\ \quad\quad \exists[\rho, \zeta_\beta, \zeta_\alpha \mid \{\rho\} * \{\zeta_\beta \mapsto \beta\} * \{\zeta_\alpha \mapsto \alpha\}]. \\ \quad\quad\quad \langle \mathcal{S}(2), \rho \texttt{ handle}, ptr(\zeta_\beta), ptr(\zeta_\alpha)\rangle \texttt{ at } \rho \end{array}$$

Every node in this list is stored in a distinct region. Therefore, when nodes are removed from list, they can be disposed of individually without having to delete the entire list. Notice that each list cell now contains a region handle to make deallocation possible.

We can define a list of lists just as easily as we did before:

$$Listof\,List'[\beta] = List'[List'[\beta]]$$

However, depending upon the context, we might choose to mix and match different list representations. If each of the internal lists has a different lifetime than the others but the cells of the internal lists have similar lifetimes then we might choose the type:

$$
\begin{array}{l}
\mathit{ListofList}''[\beta] = \\
\quad \mu\alpha.\exists[\rho \mid \{\rho\}].\langle \mathcal{S}(1), \rho\ \texttt{handle}\rangle\ \texttt{at}\ \rho \cup \\
\quad\quad \exists[\rho, \zeta_\beta, \zeta_\alpha \mid \{\rho\} * \{\zeta_\beta \mapsto \mathit{List}[\rho,\beta]\} * \{\zeta_\alpha \mapsto \alpha\}]. \\
\quad\quad\quad \langle \mathcal{S}(2), \rho\ \texttt{handle}, \mathit{ptr}(\zeta_\beta), \mathit{ptr}(\zeta_\alpha)\rangle\ \texttt{at}\ \rho
\end{array}
$$

This time, each inner list is a different region than any other inner list, but all cells in each inner list are in the same region. The real benefit of type theory is the ease with which the various type-theoretic abstractions compose with one another. In this case, alias types and regions are highly compatible and bringing them together results in a powerful language for typed memory management.

4.5.2 Related Work

Throughout this thesis I have emphasized the connections between linear typing and the novel type systems I have developed. However, there are several other formalisms for reasoning about computational effects in programming languages including type-and-effects systems [GL86, Luc87, JG91, TT94] and monads [Mog91, PJW93, LPJ95, Fil96]. Many researchers are actively investigating the relationships between these different areas, but the overall picture is not yet fully understood. I am eager to continue this line of research and explore the formal links between this system and the others.

In related work with Karl Crary and Greg Morrisett [WCM00], I have given a translation from Tofte and Talpin's region calculus into the Capability Calculus, demonstrating that the relationship between type and effect systems and capabilities is quite close. A necessary prerequisite for the use of either system is type inference, performed by a programmer or compiler, and much of the research into effects systems has concentrated on this difficult task. However, because of the focus on inference, effect systems are usually formulated as a bottom-up synthesis of effects. This work may viewed as producing verifiable evidence of the correctness of an inference. Hence, while effect systems typically work bottom-up, specifying the effects that might occur, we take a top-down approach, specifying by capabilities the effects that are *permitted to occur*. Moreover, unlike Tofte and Talpin's effect system, capabilities are sensitive to control-flow. Rather than constructing the overall effect of an expression by taking the union of the effects of the subexpressions, and thereby losing information about the order of evaluation, a type checker verifies programs are safe by checking one instruction after another and using the capability produced by previous instructions to verify that following instructions are safe. Because capabilities take evaluation order into consideration, it is possible to use capabilities to ensure that a sequence of system calls is made in the correct order or that programs follow particular security protocols. This is a fruitful area for research and it is discussed more completely in chapter 5.

A connection can also be drawn between capabilities and monadic type systems. Work relating effects to monads has viewed effectful functions as pure functions that return state transformers. This might be called an *ex post* view: the effect takes place after the function's execution. In contrast, we take an *ex ante* view in which the capability to perform the relevant effect must be satisfied *before* the function's execution. Nevertheless, there is considerable similarity between the views; just as monads can be used to ensure that the store is single-threaded

through a computation, our typing rules thread a capability (which summarizes aspects of the store) along the execution path of a program.

Region-Based Memory Management There has been much prior research on the theory and implementation of region-based memory management. With respect to implementation, Birkedal *et al.* [BTV96] describe several optimizations to the basic region-allocation scheme that are used in the ML Kit with Regions to improve space-efficiency. One of their observations is that functions can be used in two different contexts: one context in which no live object remains in a region after a function call and a second context in which there may be live objects remaining in a region after a call. In order to avoid code duplication and yet ensure efficient space usage, the call site passes information to the called function at run time. Using this information, the function may make dynamic decisions about region deallocation. The type system we present here is not powerful enough to encode these *storage-mode polymorphic* functions. However, we believe these dynamic tests may be viewed as a form of intensional type analysis [HM95, CWM98], and, therefore, if we augment the Capability Calculus with a variant of Harper and Morrisett's typecase mechanism, we may be able to verify the results of storage-mode optimizations as well.

Aiken *et al.* [AFL95] have also studied how to optimize the original Tofte-Talpin region framework. As in the Capability Calculus, they separate region allocation from region deallocation. However, they have not presented a technique for verifying that the results of their optimizations are safe. We conjecture, based on the soundness proof for Aiken *et al.*'s analyses, that the analysis could be used to produce typing annotations and that verification could take place using the Capability Calculus.

Gay and Aiken [GA98] have developed extensions to C that gives programmers complete control over region allocation and deallocation. They use reference counting to prevent programmers from accidentally accessing deallocated regions. Hawblitzel and von Eicken [HvE98] have also used the notion of a region in their language Passport to support sharing and revocation between multiple protection domains. Both of these groups use run-time checking to ensure safety and it would be interesting to investigate hybrid systems that combine features of our static type system with more dynamic systems.

Tofte and Talpin [TT97] have studied the soundness of region-based type systems at length. They use a greatest fixed-point construction and a co-inductive argument to prove the correctness of their region-inference scheme. In contrast, our formulation of the Capability Calculus allows us to use the syntactic proof techniques popularized by Wright and Felleisen [WF94]. However, despite the high-level differences between the proof techniques, there are illuminating similarities in some of the details. Most notably, Tofte and Talpin's proof involves a notion of *consistency* that relates source and target values in the region inference translation. Consistency is defined with respect to the effect (ψ) of the rest of the computation. Informally, one of the consistency conditions states that a source value is consistent with a target value in region ρ, with respect to effect ψ, if ρ does not appear in ψ. Hence, if ρ is not in the effect, or *capability*, of the rest of the computation, then we can deallocate that region because the rest of the computation cannot distinguish a dangling pointer into ρ from a value in the source language. Therefore, within the Tofte-Talpin proof, the effect of the rest of the computation plays a role very similar to a capability. We are able to give a syntactic proof of soundness for our language because continuations and their capabilities are explicit in our framework whereas Tofte and Talpin introduce this idea as a meta-level construction in their proof.

Since the Capability Calculus was developed, other research groups have investigated alter-

native proofs of Tofte and Talpin's type system. Banerjee, Heintz and Riecke [BHR99] translate a simplified version of the region calculus (without effect variables) into a variant of system F. They develop a model for the language and use it to prove deallocation is safe. Dal Zilio and Gordon [DG00] discuss the relationship between Tofte and Talpin's language and the π-calculus with groups. They also give an extremely elegant proof of soundness (again without considering effect variables). It may be a simple matter to extend these proofs to handle Tofte and Talpin's effect variables, but this point deserves scrutiny. During the initial development of my capability framework, I only considered linear and duplicatable regions $\{\rho\}$ and $\overline{\{\rho\}}$ and simple linear effect variables ϵ. However, after attempting a translation of Tofte and Talpin's full region calculus, I discovered that the non-linear type constructor had to be lifted to effect variables as in $\overline{\epsilon}$ and, more generally, to arbitrary capabilities as in $\overline{C}$.

Chapter 5

Summary and Directions for Future Research

It is difficult to make predictions,
especially about the future.

– *Yogi Berra*

In this thesis, I have developed type systems capable of encoding a variety of memory management invariants and for each type system, I have formally investigated two central meta-theoretic properties. The first property is Type Soundness. It ensures programs do not crash due to memory errors and is crucial if the type system is to be used in mission-critical software or as the basis of a security system. The second property is a garbage collection property. It prevents programs from wastefully discarding memory resources and can be used to limit certain kinds of memory leaks.

The central novelty of the more advanced type systems described in this thesis is that they view the store as an implicit additional parameter to every function and give it a type. This sort of store-passing style is quite common in Hoare-logic program correctness proofs, but I do not know of any other type systems that use the idea.[1] Given a collection of store types, it is possible to apply more standard type theoretic ideas to the problem of memory management:

- Singleton types specify object identity. They provide useful must-alias information both for single memory blocks and for entire memory regions.
- Linear types safely control state changes: An update can alter the form of a linear object and such changes can be represented in the object's type. The deallocation operation transforms a useful object into an unusable one, and again, the state transition may be represented in the type system.
- Values of intuitionistic type may be efficiently copied and shared.
- Parametric polymorphism helps increase code reuse. Location, region and store polymorphism helps hide the size and shape of the store and inconsequential details of region names and concrete locations.

[1] Haskell and some other purely functional languages encapsulate the store in a monad, which gets threaded through the program in a store-passing style. However, these monadic systems do reveal the store's structure and, consequently, are not nearly as expressive as the languages described in this thesis.

- Bounded parametric polymorphism makes it possible to restrict privileges on a particular data structure for the duration of a function call but to recover these privileges when the function returns.
- Existential types encapsulate large data structures and hide the details from the outside world.
- Recursive types specify repeating structure in the store.

5.1 Other Memory Management Strategies

There are so many different memory management strategies that I could not possibly explore them all in a single thesis.[2] As a result, there is still much work to be done this topic.

Regions And Linear Types In chapter 4, I informally developed the idea that regions can be combined with linear or alias types. Certainly, there is work to be done to fully formalize these ideas. However, perhaps more interesting is research that attempts to lift these ideas out of compiler intermediate and target languages and exposes them to source language programmers. Currently, there are no safe languages that provide explicit memory management primitives. For space-critical applications, it is almost always necessary to live dangerously with a language like C. I hope that using the advanced type systems described in this thesis, it will be possible to define a C-like language with explicit control over memory. The central challenge in this project is to find the right balance between explicit programmer annotations that give control over allocation and type inference techniques that take care of checking unimportant details and that scale effectively. It is also extremely important that the cost model be well-defined and easy to understand so that programmers can analyze and rewrite their own programs to improve performance. The Popcorn language [MCG+99] has C syntax and control-flow constructs but an ML-style typing discipline. It is an ideal starting point for such an effort. Deline and Fähndrich [DF00] are in the preliminary stages of a similar design.

Reference Counting Chirimar, Gunter and Riecke [CGR92] demonstrated that reference counting has close connections with intuitionistic linear type systems by proving that the former can safely implement the latter. One can easily imagine that a similar relationship holds between a calculus of linear and intuitionistic regions (similar to the one proposed in chapter 4) and the reference counting implementation of regions for C investigated by Gay and Aiken [GA98]. One avenue for future research is to try to use this observation to optimize Gay and Aiken's reference counting strategy by performing run-time reference counting operations at compile time instead. Static checking also detects errors earlier in the software development process. Alternatively, using dynamic reference counts may increase the flexibility of (mostly) static region-based type systems considerably. Certainly, there is a lot of room to explore the trade-offs between static and dynamic approaches to memory management.

Stack Allocation In chapter 3, I showed how to encode a stack typing discipline in the alias types framework. In the encoding, each activation record was allocated on the heap so sharing of stack frames could be handled in the same way as sharing of any other heap-allocated data structure. In earlier work with Greg Morrisett, Karl Crary and Neal Glew [MCGW98], I

[2] Thank God.

presented a typing discipline for a more conventional stack implementation. The latter typing discipline has a much more limited capacity to represent aliasing. It gives rise to two classes of stack pointers, a "strong" stack pointer that lives in a fixed register and any number of weak stack pointers that depend on it. Modifications to the strong pointer may invalidate a weak pointer under certain conditions. Moreover, pointers to the stack are always distinguished from pointers into the heap, making it impossible to encode certain stack-allocation strategies. It appears as though combining the technology described in this thesis, particularly singleton types and bounded quantification, with the earlier stack typing discipline could eliminate many, if not all, of these deficiencies.

Tracing Garbage Collection There are countless tracing garbage collection algorithms. Wilson [Wil92] provides an excellent survey of the most common ones. While a great deal is known about garbage collection when it is the only memory management technique being used, more research is required to discover how it interacts with static memory management techniques, such as those described in this thesis. Neils Hallsberg has an excellent start in this area. He has investigated the trade-offs involved in combining region inference and garbage collection [Hal99]. Wang and Appel [WA99] are attempting to write a garbage collector in a type-safe programming language with regions.

5.2 Towards More Expressive Safety Policies

The goal of this thesis is to demonstrate how to use type-theoretic technology to enforce memory safety. However, similar techniques may be used to enforce other safety policies.

Concurrency It is possible to ensure mutually-exclusive access to shared mutable data in a multi-threaded environment, by viewing locks as analogous to regions. More precisely, the set of locks currently held by a thread may be described by a type C and every function could be annotated with the lock set it requires for safe execution (just as a function in the region calculus is annotated with the live regions it requires). If we associate each piece of sensitive data with a lock, we can statically check that every client of the data obtains the corresponding lock before attempting access. When the code releases the lock, the type system would revoke the capability to dereference the data, just as it revokes the capability to dereference certain objects when a region is freed. Flanagan and Abadi [FA99] have investigated this idea in the context of a high-level lexically-scoped language. I conjecture it is possible to compile Flanagan and Abadi's locking language into a variant of my region calculus with locking primitives instead of allocation primitives.

Reliability and Security Files, windows, executable programs, CPU cycles and hardware devices such as printers or network ports are all valuable resources; a well-functioning computer system must carefully control how these components are manipulated. In an environment of cooperating processes, safe use of common resources is an important reliability issue. When a software system interacts with one or more untrusted and potentially malicious agents, the issue becomes one of security. In either case, when the stakes are high, it becomes increasingly important to be able to give strong guarantees about software quality. Once again, in these situations, it appears that type theory may be able to contribute to a solution. Static type systems detect errors early in the software development cycle, well before software is deployed in mission-critical or high-security domains.

By generalizing store types so they express a broader collection of constraints on the state of the computation and its environment, it is possible to restrict access to hardware devices and disallow dissemination of private files. It is also possible to check that untrusted software obeys particular security protocols. I have taken preliminary steps in this direction by defining a flexible, typed intermediate language capable of representing and verifying a wide range of security policies [Wal00]. Several other researchers, including Skalka and Smith [SS00] and Deline and Fähndrich [DF00] have initiated projects for analysis and verification of similar properties in high-level languages. Still others [RG94, NL97, HP99, CW00] have studied termination properties and resource bounds policies. However, the research space remains wide open. We still do not know how to integrate these techniques with mainstream programming languages for the analysis and certification of a broad, practical set of safety policies.

5.3 Conclusions

Accidentally dereferencing dangling pointers and forgetting to deallocate dead objects are amongst the most common programming errors in languages with explicit memory management. This thesis is the first step in a research program aimed at eliminating these sorts of errors through the use of flexible static type systems. It demonstrates that by combining linear and dependent typing mechanisms, it is possible to control aliasing and to enforce a variety of memory management invariants. Moreover, through the use of operational semantics and relatively straightforward syntactic proof techniques, I have obtained strong theorems about memory safety and garbage collection properties of the languages defined herein. The next step in my research program is to bring these advanced type-theoretic techniques to bear on main-stream source-level programming languages. Careful consideration of type structure and programming language design can have a profound impact on the reliability, integrity and security of practical software systems.

Appendix A

Alias Type Soundness & Garbage Collection Properties

The proof of aliasing type soundness is the first write-only appendix of this thesis.

Lemma 21 *If* $\Delta \vdash_A c : \kappa$ *then* $\mathrm{FV}(c) \subseteq \mathit{Dom}(\Delta)$.

Proof: By induction on the kinding and equality derivations.

Lemma 22 *If* $\Delta \vdash_A c : \kappa$ *and* $\alpha \notin \mathit{Dom}(\Delta)$ *then* $\Delta, \alpha{:}\kappa' \vdash_A c : \kappa$

Proof: By induction on the kinding derivation.

Lemma 23 (Type Substitution) *If* $\cdot \vdash_A c : \kappa$ *and* $\theta = [c/\alpha]$ *then*

1. *if* $\Delta, \alpha{:}\kappa, \Delta'; \Gamma \vdash_A v : \tau$ *then* $\Delta, \Delta'; \theta(\Gamma) \vdash_A \theta(v) : \theta(\tau)$.
2. *if* $\Delta, \alpha{:}\kappa, \Delta'; \Gamma; C \vdash_A e$ *then* $\Delta, \Delta'; \theta(\Gamma); \theta(C) \vdash_A \theta(e)$

Proof: By induction on the typing derivation.

Lemma 24 (Value Substitution) *If* $\cdot; \cdot \vdash_A v : \tau$ *and* $\theta = [v/x]$ *then*

1. *if* $\Delta; \Gamma, x{:}\tau, \Gamma' \vdash_A v' : \tau'$ *then* $\Delta; \Gamma, \Gamma' \vdash_A \theta(v') : \tau'$
2. *if* $\Delta; \Gamma, x{:}\tau, \Gamma'; C \vdash_A e$ *then* $\Delta; \Gamma, \Gamma'; C \vdash_A \theta(e)$

Proof: By induction on the typing derivation.

Lemma 25 *If for all* $x \in \mathit{Dom}(\Gamma)$, $\Delta \vdash_A \Gamma(x) : \mathtt{Type}$ *and* $\Delta; \Gamma \vdash_A v : \tau$ *and* $\Delta; \Gamma \vdash_A v : \tau'$ *then* $\Delta \vdash_A \tau = \tau' : \mathtt{Type}$

Proof: By induction on the typing derivation.

Lemma 26 *If for all* $x \in \mathit{Dom}(\Gamma), \Delta \vdash_A \Gamma(x) = \Gamma'(x) : \mathtt{Type}$ *and* $\Delta \vdash_A C = C' : \mathtt{Store}$ *then*

1. *if* $\Delta; \Gamma \vdash_A v : \tau$ *then* $\Delta; \Gamma' \vdash_A v : \tau$
2. *if* $\Delta; \Gamma; C \vdash_A e$ *then* $\Delta; \Gamma'; C' \vdash_A e$

Proof: By induction on the typing derivation.

Lemma 27 (Canonical Small Value Forms) *If* $\cdot;\cdot \vdash_A v : \tau$ *then*

1. *if* $\tau = \mathcal{S}(i)$ *then* $v = \mathcal{S}(i)$
2. *if* $\tau = ptr(\ell)$ *then* $v = \mathtt{ptr}(\ell)$
3. *if* $\tau = \forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}$ *then*
   (a) $v = v'[c'_1, \ldots, c'_n]$ *and*
   $v' = \mathtt{fix}\ f\ [\Delta', \Delta](C', x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n).e$
   (b) $\Delta' = \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$
   (c) *for* $1 \leq i \leq n$, $\cdot \vdash_A c'_i : \kappa_i$
   (d) $\cdot \vdash_A \mathtt{fix}\ f\ [\Delta', \Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \tau$
   (e) $\cdot \vdash_A (\forall[\Delta].(C', \tau'_1, \ldots, \tau'_n) \to \mathbf{0})[c_1, \ldots, c_n/\Delta'] = (\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}) :$

Proof: By induction on the typing derivation.

Lemma 28 (Canonical Storable Value Forms) *If* $\vdash_A h : \tau$ *then*

1. *if* $\tau = \langle \tau_1, \ldots, \tau_n \rangle$ *then*
   (a) $h = \langle v_1, \ldots, v_n \rangle$
   (b) $\vdash_A h_i : \tau_i$
2. *if* $\tau = \tau_1 \cup \tau_2$ *then*
   (a) $h = \mathtt{union}_{\tau'_1 \cup \tau'_2}(h')$
   (b) $\vdash_A h' : \tau_i$ *where* $i = 1$ *or* 2
   (c) $\cdot \vdash_A \tau_1 = \tau'_1 : \mathtt{Type}$
   (d) $\cdot \vdash_A \tau_2 = \tau'_2 : \mathtt{Type}$
3. *if* $\tau = \mathtt{rec}\,\alpha\,(\Delta).\tau'\,(c_1, \ldots, c_n)$ *then*
   (a) $h = \mathtt{roll}_{\tau''}(h')$
   (b) $\vdash_A h' : \tau'[\mathtt{rec}\,\alpha\,(\Delta).\tau'/\alpha][c_1, \ldots, c_n/\Delta]$
   (c) $\cdot \vdash_A \tau = \tau'' : \mathtt{Type}$
4. *if* $\tau = \exists[\Delta \mid C].\tau'$ *then*
   (a) $h = \mathtt{pack}_{[c_1, \ldots, c_n \mid S]\mathtt{as}\tau''}(h')$
   (b) $\cdot \vdash_A \tau = \tau'' : \mathtt{Type}$
   (c) $\Delta = \alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n$
   (d) *for* $1 \leq i \leq n$, $\cdot \vdash_A c_i : \kappa_i$
   (e) $\vdash_A h' : \tau'[c_1, \ldots, c_n/\Delta]$
   (f) $\vdash_A S : C[c_1, \ldots, c_n/\Delta]$

Proof: By induction on the typing derivation.

Lemma 29 *$\cdot \vdash_A \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m\} = \{\ell'_1 \mapsto \tau'_1, \ldots, \ell'_n \mapsto \tau'_n\} : \texttt{Store}$ if and only if*

1. *$n = m$ and for some one-to-one function $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$, $\ell_{\pi(1)}, \ldots, \ell_{\pi(m)} = \ell'_1, \ldots, \ell'_m$*
2. *for $1 \leq i \leq m$, $\cdot \vdash_A \tau_{\pi(i)} = \tau'_i : \texttt{Type}$*

Proof:

($\Rightarrow$) By induction on the store type equality judgement.
($\Leftarrow$) By successive application of the rule `AT-exchange`, we have:

$$\cdot \vdash_A \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m\} = \{\ell_{\pi(1)} \mapsto \tau_{\pi(1)}, \ldots, \ell_{\pi(m)} \mapsto \tau_{\pi(m)}\} : \texttt{Store}$$

Now, for $1 \leq i \leq m$, $\cdot \vdash_A \tau_{\pi(i)} = \tau'_i : \texttt{Type}$ therefore, by the congruence rule for store types:

$$\frac{\cdot \vdash_A \tau_{\pi(i)} = \tau'_i : \texttt{Type} \quad \text{for } 1 \leq i \leq m}{\cdot \vdash_A \{\ell_{\pi(1)} \mapsto \tau_{\pi(1)}, \ldots, \ell_{\pi(m)} \mapsto \tau_{\pi(m)}\} = \{\ell_1 \mapsto \tau'_1, \ldots, \ell_m \mapsto \tau'_m\} : \texttt{Store}}$$

By transitivity of equality,

$$\cdot \vdash_A \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m\} = \{\ell'_1 \mapsto \tau'_1, \ldots, \ell'_n \mapsto \tau'_n\} : \texttt{Store}$$

□

Lemma 30 (Canonical Store Forms) *If $\vdash_A S : C$ and $\cdot \vdash_A C = C' * \{\ell'_m \mapsto \tau'_m\} : \texttt{Store}$ then*

1. *$S = S'\{\ell'_m \mapsto h\}$*
2. *$\vdash_A S' : C'$*
3. *$\vdash_A h : \tau'_m$*

Proof:

The typing derivation $\vdash_A S : C$ has the form

$$\frac{\begin{array}{c} S = \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\ \cdot \vdash_A C = \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\} : \texttt{Store} \\ \vdash_A h_i : \tau_i \qquad (\text{for } 1 \leq i \leq n) \end{array}}{\vdash_A S : C}$$

From the fact that $\cdot \vdash_A C = C' * \{\ell'_m \mapsto \tau'_m\} : \texttt{Store}$, and by Lemma 11 and Lemma 21, C' can contain no free variables and therefore may be written $\{\ell'_1 \mapsto \tau'_1, \ldots, \ell'_{m-1} \mapsto \tau'_{m-1}\}$. By symmetry and transitivity of equality,

$$\cdot \vdash_A \{\ell'_1 \mapsto \tau'_1, \ldots, \ell'_{m-1} \mapsto \tau'_{m-1}, \ell'_m \mapsto \tau'_m\} = \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\} : \texttt{Store}$$

By lemma 29, we know that

1. $n = m$ and for some one-to-one function $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$, $\ell_{\pi(1)}, \ldots, \ell_{\pi(m)} = \ell'_1, \ldots, \ell'_m$

2. for $1 \leq i \leq m$, $\cdot \vdash_A \tau_{\pi(i)} = \tau'_i : \texttt{Type}$

Hence,

3 $n = m$ and for some one-to-one function $\pi : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\}$, $\ell_{\pi(1)}, \ldots, \ell_{\pi(m-1)} = \ell'_1, \ldots, \ell'_{m-1}$

4 for $1 \leq i \leq m-1$, $\cdot \vdash_A \tau_{\pi(i)} = \tau'_i : \texttt{Type}$

Consequently, assuming $\pi(m) = k$, by lemma 29, we know that

$$\cdot \vdash_A C' = \{\ell_1 \mapsto \tau_1, \ldots, \ell_{k-1} \mapsto \tau_{k-1}, \ell_{k+1} \mapsto \tau_{k+1}, \ldots, \ell_m \mapsto \tau_m\} : \texttt{Store}$$

Let S' be $\{\ell_1 \mapsto h_1, \ldots, \ell_{k-1} \mapsto h_{k-1}, \ell_{k+1} \mapsto h_{k+1}, \ldots, \ell_m \mapsto h_m\}$. We have $S = S'\{\ell_k \mapsto h_k\}$. We also know from the judgement $\vdash_A C : S$ that for $1 \leq i \leq m$, $i \neq k$, $\vdash_A h_i : \tau_i$. Therefore,

$$\frac{\begin{array}{c} S' = \{\ell_1 \mapsto h_1, \ldots, \ell_{k-1} \mapsto h_{k-1}, \ell_{k+1} \mapsto h_{k+1}, \ldots, \ell_m \mapsto h_m\} \\ \cdot \vdash_A C' = \{\ell_1 \mapsto \tau_1, \ldots, \ell_{k-1} \mapsto \tau_{k-1}, \ell_{k+1} \mapsto \tau_{k+1}, \ldots, \ell_m \mapsto \tau_m\} : \texttt{Store} \\ \vdash_A h_i : \tau_i \qquad (\text{for } 1 \leq i \leq m, i \neq k) \end{array}}{\vdash_A S' : C'}$$

From the judgement $\vdash_A S : C$, we also know that $\vdash_A h_k : \tau_k$ and since $\cdot \vdash_A \tau_k = \tau'_m : \texttt{Type}$, we have $\vdash_A h : \tau'_m$ using the rule **A-veq**.

□

Lemma 31 *If* $\vdash_A S'\{\ell \mapsto h\} : C$ *and* $\cdot \vdash_A C = C' * \{\ell \mapsto \tau\} : \texttt{Store}$ *then*

1. $\vdash_A S' : C'$

2. $\vdash_A h : \tau$

Proof: Corollary of Canonical Store Forms.

Lemma 32 *If*

1. $\vdash_A S'\{\ell \mapsto h\} : C$

2. $\cdot \vdash_A C = C' * \{\ell \mapsto \tau\} : \texttt{Store}$

3. $\vdash_A h' : \tau'$

then $\vdash_A S'\{\ell \mapsto h'\} : C' * \{\ell \mapsto \tau'\}$

Proof:

By (1), (2), and Lemma 31, $\vdash_A S' : C'$. Hence, by (3) and inspection of the store typing judgement, $\vdash_A S'\{\ell \mapsto h'\} : C' * \{\ell \mapsto \tau'\}$

□

Lemma 33 *If* $Dom(S) \cap Dom(S') = \emptyset$ *and* $\vdash_A S : C$ *and* $\vdash_A S' : C'$ *then* $\vdash_A SS' : C * C'$

Proof:

Since $Dom(S) \cap Dom(S') = \emptyset$, we know that SS' is a store. Inspection of the store typing judgement and examination of the definition of equality provides the result.

□

Lemma 34

1. *If* $\Delta; \Gamma \vdash_A v : \tau$ *then* $\mathtt{L}(v) = 0$.
2. *If* $\Delta; \Gamma; C \vdash_A e$ *then* $\mathtt{L}(e) = 0$.

Proof: By induction on the typing derivations.

Lemma 35 (Preservation) *If* $\vdash_A (S, e)$ *and* $(S, e) \longmapsto_A (S', e')$ *then* $\vdash_A (S', e')$

Proof:

By cases on the operational rule that applied.

- Operational Rule:

$$(S, \mathtt{let}\,\zeta, x = \mathtt{new}\ (i)\ \mathtt{in}\ e) \longmapsto_A (S\{\ell \mapsto h\}, e[\ell/\zeta][\mathtt{ptr}(\ell)/x])$$

  where $\ell \notin S, e$ and $h = \overbrace{\langle _, \ldots, _ \rangle}^{i}$
  Typing Assumption:
  1. $\mathtt{GU}(S)$
  2. $\vdash S : C$.
  3. $\cdot;\cdot; C \vdash_A \mathtt{let}\,\zeta, x = \mathtt{new}\ (i)\ \mathtt{in}\ e$.

  First, we know $\mathtt{GU}(S\{\ell \mapsto h\})$ since $\mathtt{GU}(S)$ and $\ell \notin S$.
  Second, we can conclude:

$$\vdash_A h : \overbrace{\langle \mathit{junk}, \ldots, \mathit{junk} \rangle}^{i}$$

  Consequently, by inspection of the store typing judgement,

$$\vdash_A S\{\ell \mapsto h\} : C * \{\ell \mapsto \overbrace{\langle \mathit{junk}, \ldots, \mathit{junk} \rangle}^{i}\}$$

  Third, the premise of the instruction typing judgement is

$$\zeta{:}\mathtt{Loc}; x{:}\mathit{ptr}(\zeta); C * \{\zeta \mapsto \overbrace{\langle \mathit{junk}, \ldots, \mathit{junk} \rangle}^{i}\} \vdash_A e$$

  By Type and Value Substitution Lemmas,

$$\cdot;\cdot; C * \{\ell \mapsto \overbrace{\langle \mathit{junk}, \ldots, \mathit{junk} \rangle}^{i}\} \vdash_A e[\ell/\zeta][\mathit{ptr}(\ell)/x]$$

  Consequently, $\vdash_A (S\{\ell \mapsto h\}, e[\ell/\zeta][\mathtt{ptr}(\ell)/x])$

- Operational Rule:

$$(S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\}, \texttt{free ptr}(\ell); e) \longmapsto_A (S, e)$$

Typing Assumption:

1. $\texttt{GU}(S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\})$
2. $\vdash_A S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\} : C$.
3. $\cdot; \cdot; C \vdash_A \texttt{free ptr}(\ell); e$.

From 1, we have $\texttt{GU}(S)$.
The instruction judgement 3 must have the form:

$$\frac{\begin{array}{c}\cdot; \cdot \vdash_A v : ptr(\ell) \\ \cdot \vdash_A C = C' * \{\ell \mapsto \langle \tau_1, \dots, \tau_n\rangle\} : \texttt{Store} \\ \cdot; \cdot; C' \vdash_A e\end{array}}{\cdot; \cdot; C \vdash_A \texttt{free } v; e}$$

Hence, by Lemma 31, we can deduce that $\vdash_A S : C'$.
Moreover, we have $\cdot; \cdot; C' \vdash_A e$ directly.
Therefore, $\vdash_A (S, e)$.

- Operational Rule:

$$\begin{array}{l}(S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\}, \texttt{let } x = \texttt{ptr}(\ell).i \texttt{ in } e) \longmapsto_A \\ \quad (S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\}, e[v_i/x])\end{array}$$

where $1 \leq i \leq n$
Typing Assumption:

1. $\texttt{GU}(S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\})$
2. $\vdash_A S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\} : C$.
3. $\cdot; \cdot; C \vdash_A \texttt{let } x = \texttt{ptr}(\ell).i \texttt{ in } e$.

Where the instruction derivation has the following form:

$$\frac{\begin{array}{c}\cdot; \cdot \vdash_A \texttt{ptr}(\ell) : ptr(\ell) \\ \cdot \vdash_A C = C' * \{\ell \mapsto \langle \tau_1, \dots, \tau_n\rangle\} : \texttt{Store} \\ \cdot; x{:}\tau_i; C \vdash_A e\end{array}}{\cdot; \cdot; C \vdash_A \texttt{let } x = \texttt{ptr}(\ell).i \texttt{ in } e}$$

The instruction derivation contains the fact $\cdot; x{:}\tau_i; C \vdash_A e$ and by Lemma 31, we can conclude that $\cdot; \cdot \vdash_A \langle v_1, \dots, v_n\rangle : \langle \tau_1, \dots, \tau_n\rangle$. By Canonical Storable Value Forms, we can conclude that $\cdot; \cdot \vdash_A v_i : \tau_i$. Hence, from the Value Substitution Lemma, we know that $\cdot; \cdot; C \vdash_A e[v_i/x]$. From this judgement and (1) and (2), we can conclude $\vdash_A (S\{\ell \mapsto \langle v_1, \dots, v_n\rangle\}, e[v_i/x])$

- Operational Rule:

$$\begin{array}{l}(S\{\ell \mapsto \langle v_1, \dots, v_i, \dots, v_n\rangle\}, \texttt{ptr}(\ell).i{:=}v'; e) \longmapsto_A \\ \quad (S\{\ell \mapsto \langle v_1, \dots, v', \dots, v_n\rangle\}, e)\end{array}$$

Typing Assumption:

1. $\texttt{GU}(S\{\ell \mapsto \langle v_1, \ldots, v_i, \ldots, v_n \rangle\})$
2. $\vdash_A S\{\ell \mapsto \langle v_1, \ldots, v_i, \ldots, v_n \rangle\} : C.$
3. $\cdot; \cdot; C \vdash_A \mathtt{ptr}(\ell).i{:=}v'; e.$

The instruction typing judgement has the form:

$$\frac{\begin{array}{c} \cdot; \cdot \vdash_A \mathtt{ptr}(\ell) : ptr(\ell) \\ \cdot; \cdot \vdash_A v' : \tau \\ \cdot \vdash_A C = C' * \{\ell \mapsto \langle \tau_1, \ldots, \tau_i, \ldots, \tau_n \rangle\} : \mathtt{Store} \\ \cdot; \cdot; C' * \{\ell \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\} \vdash_A e \end{array}}{\cdot; \cdot; C \vdash_A \mathtt{ptr}(\ell).i{:=}v'; e}$$

By Lemma 31 we can conclude that

$$\vdash_A \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle$$

By Canonical Storable Value Forms, we can deduce that (4) for all $1 \leq i \leq n$

$$\cdot; \cdot \vdash_A v_j : \tau_j$$

From the instruction typing judgement, we also have the fact that (5)

$$\cdot; \cdot \vdash_A v' : \tau$$

By Lemma 34, $\texttt{L}(v_i) = \texttt{L}(v') = 0$.
Consequently, since $\texttt{GU}(S\{\ell \mapsto \langle v_1, \ldots, v_i, \ldots, v_n \rangle\})$, we know (6)

$$\texttt{GU}(S\{\ell \mapsto \langle v_1, \ldots, v', \ldots, v_n \rangle\})$$

By Lemma 31 we can also conclude that $\vdash_A S : C'$. From (4), (5), and the typing rule for tuples, we can conclude that (7)

$$\cdot; \cdot \vdash_A \langle v_1, \ldots, v', \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle$$

Therefore, from (2), (7), and the constraint equality judgement from the instruction typing judgement, we can conclude by Lemma 32 that

$$\vdash_A S\{\ell \mapsto \langle v_1, \ldots, v', \ldots, v_n \rangle\} : C' * \{\ell \mapsto \langle \tau_1, \ldots, \tau, \ldots, \tau_n \rangle\}$$

These facts and the instruction typing judgement imply that

$$\vdash_A (S\{\ell \mapsto \langle v_1, \ldots, v', \ldots, v_n \rangle\}, e)$$

- Operational Rule

$$(S\{\ell \mapsto h\}, \mathtt{if}\ \mathtt{ptr}(\ell)\ (e_1 \mid e_2)) \longmapsto_A (S\{\ell \mapsto h'\}, e_i)$$

where

$$\begin{array}{rcl} i & = & 1 \text{ or } 2 \\ h & = & \mathtt{union}_{\tau_1 \cup \tau_2}(h') \\ h' & = & \varsigma_1(\cdots \varsigma_m(\langle \mathcal{S}(i), v_1, \ldots, v_n \rangle) \cdots) \end{array}$$

Typing Assumption:

1. $\texttt{GU}(S\{\ell \mapsto h\})$
2. $\vdash_A S\{\ell \mapsto h\} : C$.
3. $\cdot;\cdot;C \vdash_A \texttt{if ptr}(\ell)\ (e_1 \mid e_2)$.

Where the instruction typing judgement has the form:

$$\frac{\begin{array}{c}\cdot \vdash_A \texttt{ptr}(\ell) : ptr(\ell) \\ \cdot \vdash_A C = C' * \{\ell \mapsto \tau_1 \cup \tau_2\} : \texttt{Store} \\ \cdot \vdash_A \tau_1 = \tau_1' : \texttt{Type} \\ \cdot \vdash_A \tau_2 = \tau_2' : \texttt{Type} \\ \cdot;\cdot;C' * \{\ell \mapsto \tau_1\} \vdash_A e_1 \\ \cdot;\cdot;C' * \{\ell \mapsto \tau_2\} \vdash_A e_2\end{array}}{\cdot;\cdot;C \vdash_A \texttt{if ptr}(\ell)\ (e_1 \mid e_2)}$$

Where τ_1' is

$$\exists[\Delta_1' \mid C_1'].\cdots\exists[\Delta_j' \mid C_j'].\langle \mathcal{S}(1), \tau_1', \ldots, \tau_k' \rangle$$

and τ_2' is

$$\exists[\Delta_1'' \mid C_1''].\cdots\exists[\Delta_m'' \mid C_m''].\langle \mathcal{S}(2), \tau_1'', \ldots, \tau_n'' \rangle$$

First, by inspection of the definition of $\texttt{L}(\cdot)$, we can conclude that $\texttt{L}(h') = \texttt{L}(h)$ and therefore that $\texttt{L}(S\{\ell \mapsto h'\}) = \texttt{L}(S\{\ell \mapsto h\})$. Since $\texttt{GU}(S\{\ell \mapsto h\})$ we can conclude $\texttt{GU}(S\{\ell \mapsto h'\})$.

Second, from (2) and by Canonical Store Forms and Canonical Value Forms, we can conclude that

$$\vdash_A h' : \tau_i$$

Hence, by Lemma 32, we can deduce that

$$\vdash_A S\{\ell \mapsto h'\} : C' * \{\ell \mapsto \tau_i\}$$

From the instruction typing judgement, we have

$$\cdot;\cdot;C' * \{\ell \mapsto \tau_i\} \vdash_A e_i$$

Therefore, we can conclude that $\vdash_A (S\{\ell \mapsto h'\}, e_i)$.

- Operational Rule

$$(S, v(v_1, \ldots, v_n)) \longmapsto_A (S, \theta(e))$$

where

$$\begin{array}{rcl} v & = & v'[c_1, \ldots, c_m] \\ v' & = & \texttt{fix}\ f\ [\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ \theta & = & [c_1, \ldots, c_m/\Delta][v'/f][v_1, \ldots, v_n/x_1, \ldots, x_n] \\ \Delta & = & \beta_1{:}\kappa_1, \ldots, \beta_m{:}\kappa_m \end{array}$$

Typing Assumption:

1. $\texttt{GU}(S)$
2. $\vdash_A S : C$.
3. $\cdot;\cdot;C \vdash_A v(v_1, \ldots, v_n)$.

Where the instruction typing judgement has the form:

$$\frac{\cdot;\cdot \vdash_A v : \forall[\cdot].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0} \quad \cdot;\cdot \vdash_A v_1 : \tau_1 \quad \cdots \quad \cdot;\cdot \vdash_A v_n : \tau_n}{\cdot;\cdot; C \vdash_A v(v_1, \ldots, v_n)}$$

By Canonical Small Value Forms,

$$\Delta; f{:}\forall[\Delta].(C', \tau_1', \ldots, \tau_n') \to \mathbf{0}, x_1{:}\tau_1', \ldots, x_n{:}\tau_n'; C' \vdash_A e$$

and

$$\cdot \vdash_A \theta(\forall[\cdot].(C', \tau_1', \ldots, \tau_n') \to \mathbf{0}) = \forall[\cdot].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0} : \mathtt{Type}$$

By definition of equality $\cdot \vdash_A \theta(C') = C : \mathtt{Store}$ and also, for $1 \leq i \leq m$,

$$\cdot \vdash_A c_i : \kappa_i$$

From the instruction typing judgement, each value v_i is well-formed with type τ_i. Consequently, by Type and Value Substitution, we know that

$$\cdot;\cdot;\theta(C') \vdash_A \theta(e)$$

By Lemma 26 and the fact that $\cdot \vdash_A \theta(C') = C : \mathtt{Store}$, we have

$$\cdot;\cdot; C \vdash_A \theta(e)$$

From this fact (1) and (2), we can conclude $\vdash_A (S, \theta(e))$.

- Operational Rule

$$(S, \mathtt{coerce}(\gamma); e) \longmapsto_A (S', \theta(e))$$

where $\gamma(S) \longmapsto_A S', \theta$ and $\gamma = \mathtt{union}_{\tau_1 \cup \tau_2}(\ell)$ or $\mathtt{roll}_\tau(\ell)$ or $\mathtt{unroll}(\ell)$ is similar to the case for the instruction $v.i{:=}v'; e$.

- Operational Rule

$$(S\{\ell \mapsto h\}, \mathtt{coerce}(\gamma); e) \longmapsto_A (SS'\{\ell \mapsto h'\}, \theta(e))$$

where

$$\begin{array}{rcl} h & = & \mathtt{pack}_{[c_1,\ldots,c_n|S']\mathtt{as}\exists[\Delta|C].\tau}(h') \\ \gamma & = & \mathtt{unpack}\,\ell\,\mathtt{with}\,\Delta \\ \theta & = & [c_1, \ldots, c_n/\Delta] \\ \Delta & = & \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \end{array}$$

Typing Assumption:

1. $\mathtt{GU}(S\{\ell \mapsto h\})$
2. $\vdash_A S\{\ell \mapsto h\} : C$.
3. $\cdot;\cdot; C \vdash_A \mathtt{coerce}(\gamma); e$.

Where the coercion typing judgement is

$$\frac{\cdot \vdash_A C = C'' * \{\ell \mapsto \exists[\Delta \mid C'].\tau\} : \mathtt{Store}}{\cdot; C \vdash_A \mathtt{unpack}\,\ell\,\mathtt{with}\,\Delta \implies \Delta; C'' * \{\ell \mapsto \tau\} * C'}$$

and the instruction typing judgement implies e is well-formed:

$$\Delta; \cdot; C'' * \{\ell \mapsto \tau\} * C' \vdash_A e$$

By inspection of the definition of $\mathtt{L}(\cdot)$, we can deduce that $\mathtt{L}(S\{\ell \mapsto h\}) = \mathtt{L}(SS'\{\ell \mapsto h'\})$ Therefore, since

$$\mathtt{GU}(S\{\ell \mapsto h\})$$

, we can conclude that

$$\mathtt{GU}(SS'\{\ell \mapsto h'\})$$

By (2), the coercion typing judgement and Lemma 31 we can deduce that $\vdash_A h : \exists[\Delta \mid C'].\tau$. Consequently, by Canonical Storable Value Forms, we can conclude that $\vdash_A h' : \theta(\tau)$. From this fact, (2), and the equality from the coercion typing judgement, using Lemma 32, we can conclude that

$$\vdash_A S\{\ell \mapsto h'\} : C'' * \{\ell \mapsto \theta(\tau)\}$$

By Canonical Storable Value forms, we also know that

$$\vdash_A S' : \theta(C')$$

Therefore, by Lemma 33 we can conclude that

$$\vdash_A S\{\ell \mapsto h'\}S' : C'' * \{\ell \mapsto \theta(\tau)\} * \theta(C')$$

Hence the equivalent store is also well-formed:

$$\vdash_A SS'\{\ell \mapsto h'\} : C'' * \{\ell \mapsto \theta(\tau)\} * \theta(C')$$

Canonical Storable Value Forms also implies that for $1 \leq i \leq n$,

$$\cdot \vdash_A c_i : \kappa_i$$

Now by the instruction typing judgement and the Type Substitution Lemma we can deduce that

$$\cdot; \cdot; C'' * \{\ell \mapsto \theta(\tau)\} * \theta(C') \vdash_A \theta(e)$$

Consequently, $\vdash_A (SS'\{\ell \mapsto h'\}, \theta(e))$.

- Operational Rule:

$$(SS'\{\ell \mapsto h\}, \mathtt{coerce}(\gamma); e) \longmapsto_A (S\{\ell \mapsto h'\}, e)$$

where

$$\begin{array}{lcl} h' & = & \mathtt{pack}_{[c_1,\ldots,c_n|S']\mathtt{as}\exists[\Delta|C'].\tau}(h) \\ \gamma & = & \mathtt{pack}_{[c_1,\ldots,c_n|\theta(C')]\mathtt{as}\ \exists[\Delta|C'].\tau}(\ell) \\ \theta & = & [c_1,\ldots,c_n/\Delta] \\ S' & = & \{\ell_1 \mapsto h_1,\ldots,\ell_m \mapsto h_m\} \\ \theta(C') & = & \{\ell_1 \mapsto \tau_1,\ldots,\ell_m \mapsto \tau_m\} \end{array}$$

Typing Assumption:

1. $\mathtt{GU}(SS'\{\ell \mapsto h\})$

2. $\vdash_A SS'\{\ell \mapsto h\} : C$.
3. $\cdot; \cdot; C \vdash_A \texttt{coerce}(\gamma); e$.

Where the coercion typing judgement is

$$\frac{\Delta = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \qquad \cdot \vdash_A c_i : \kappa_i \quad (\text{for } 1 \le i \le n) \\ \cdot \vdash_A C = C'' * \{\ell \mapsto \theta(\tau)\} * \theta(C') : \texttt{Store}}{\cdot; C \vdash_A \texttt{pack}_{[c_1,\ldots,c_n|\theta(C')]\texttt{as}\,\exists[\Delta|C'].\tau}(\ell) \Longrightarrow \\ \cdot; C'' * \{\ell \mapsto \exists[\Delta \mid C'].\tau\}}$$

and the instruction typing judgement implies e is well-formed:

$$\cdot; \cdot; C'' * \{\ell \mapsto \exists[\Delta \mid C'].\tau\} \vdash_A e$$

By inspection of the definition of $\texttt{L}(\cdot)$, we can deduce that $\texttt{L}(SS'\{\ell \mapsto h\}) = \texttt{L}(S\{\ell \mapsto h'\})$. Therefore, since $\texttt{GU}(SS'\{\ell \mapsto h\})$, we can conclude that (4) $\texttt{GU}(S\{\ell \mapsto h'\})$.
From the definitions above, we know that

$$SS'\{\ell \mapsto h\} = S\{\ell_1 \mapsto h_1, \ldots, \ell_m \mapsto h_m, \ell \mapsto h\}$$

Moreover, the constraint equality judgement in the coercion typing judgement with the definition of $\theta(C')$ implies that

$$\cdot \vdash_A C = C'' * \{\ell_1 \mapsto \tau_1, \ldots, \ell_m \mapsto \tau_m, \ell \mapsto \theta(\tau)\} : \texttt{Store}$$

Now from (2) and by induction on m and Lemma 31, we can conclude the following facts:

6. $\vdash_A h : \theta(\tau)$
7. for $1 \le i \le m$, $\vdash_A h_i : \tau_i$
8. $\vdash_A S : C''$

From (7), we can prove $\vdash_A S' : \theta(C')$ using the store typing rule. Consequently, we can show that h' is well-formed:

$$\frac{\Delta = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_n \qquad \cdot \vdash_A c_i : \kappa_i \quad (\text{for } 1 \le i \le n) \\ \vdash_A S' : \theta(C') \qquad \vdash_A h : \theta(\tau)}{\vdash_A \texttt{pack}_{[c_1,\ldots,c_n|S']\texttt{as}\exists[\Delta|C'].\tau}(h) : \exists[\Delta \mid C'].\tau}$$

Since $\vdash_A S : C''$ and $\vdash_A h' : \exists[\Delta \mid C'].\tau$, we can conclude by the store typing judgement that (8)

$$\vdash_A S\{\ell \mapsto h'\} : C'' * \{\ell \mapsto \exists[\Delta \mid C'].\tau\}$$

Finally, using (4), (5), (8), we can conclude that $\vdash_A (S\{\ell \mapsto h'\}, e)$.

□

Lemma 36 (Progress) *If* $\vdash_A (S, e)$ *then either* $e = \texttt{halt}\ i$ *or* $(S, e) \longmapsto_A (S', e')$.

Proof: By cases on the syntax of the first instruction of e. The proof makes heavy use of the Canonical Forms lemmas. Here, we show a couple of representative cases.

- $(S, \texttt{let}\,\zeta, x = \texttt{new}\ (i)\ \texttt{in}\ ; e)$. This program always steps to

$$(S\{\ell \mapsto \overbrace{\langle _, \ldots, _ \rangle}^{i}\}, e[\ell/\zeta][\texttt{ptr}(\ell)/x])$$

  where ℓ is some location not present in S.

- $(S, \texttt{free}\ v; e)$.

  Typing Assumption:

  1. $\texttt{GU}(S)$
  2. $\vdash_A S : C$.
  3. $\cdot; \cdot; C \vdash_A \texttt{free}\ v; e$.

  The instruction typing judgement must have the form:

$$\frac{\begin{array}{c}\cdot; \cdot \vdash_A v : ptr(\ell) \\ \cdot \vdash_A C = C' * \{\ell \mapsto \langle \tau_1, \ldots, \tau_n \rangle\} : \texttt{Store} \\ \cdot; \cdot; C' \vdash_A e\end{array}}{\cdot; \cdot; C \vdash_A \texttt{free}\ v; e}$$

  By Canonical Small Value Forms, v must be $\texttt{ptr}(\ell)$. By Canonical Store Forms, $S = S'\{\ell \mapsto h'\}$ and $\vdash_A h' : \langle \tau_1, \ldots, \tau_n \rangle$. By Canonical Storable Value Forms again, h' must be $\langle v_1, \ldots, v_n \rangle$. Therefore this program meets the precondition for the operational rule for $\texttt{free}$ and it steps to (S', e).

- $\texttt{halt}\ v$. By inspection of the typing rule for halt, we see that

$$\frac{\cdot; \cdot \vdash_A v : ptr(\ell) \quad \cdot \vdash_A C = C' * \{\ell \mapsto \langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle\} : \texttt{Store}}{\cdot; \cdot; C \vdash_A \texttt{halt}\ v}$$

  By Canonical Small Value Forms, v must be $\texttt{ptr}(\ell)$. By Canonical Store Forms, $S = \{\ell \mapsto h'\}$. By Canonical Storable Value Forms

$$h' = \texttt{union}_{\langle \mathcal{S}(1) \rangle \cup \langle \mathcal{S}(2) \rangle}(\langle \mathcal{S}(i) \rangle)$$

  Therefore, $(S, \texttt{halt}\ v)$ is a well-formed terminal state (not stuck).

- $(S, \texttt{coerce}(\gamma); e)$ where $\gamma = \texttt{unpack}\,\ell\,\texttt{with}\,\Delta$

  Typing Assumption:

  1. $\texttt{GU}(S)$
  2. $\vdash_A S : C$.
  3. $\cdot; \cdot; C \vdash_A \texttt{coerce}(\gamma); e$.

  Where the coercion typing judgement has the form:

$$\frac{\cdot \vdash_A C = C'' * \{\ell \mapsto \exists[\Delta \mid C'].\tau\} : \texttt{Store}}{\cdot; C \vdash_A \texttt{unpack}\,\ell\,\texttt{with}\,\Delta \Longrightarrow \Delta; C'' * \{\ell \mapsto \tau\} * C'}$$

and the instruction typing judgement implies e is well-formed:

$$\Delta; \cdot; C'' * \{\ell \mapsto \tau\} * C' \vdash_A e$$

By Canonical Store Forms, $S = S'\{\ell \mapsto h\}$ and

$$\vdash_A h : \exists[\Delta \mid C'].\tau$$

By Canonical Value Forms,

$$h = \mathtt{pack}_{[c_1,\ldots,c_n \mid S'']\mathtt{as}\exists[\Delta|C'].\tau}(h')$$

By inspection of the definition of $\mathtt{GU}(S)$, we can conclude that

$$(Dom(S') \cup \{\ell\}) \cap Dom(S'') = \emptyset$$

Therefore, $S'S''\{\ell \mapsto h\}$ is a store. Canonical Value Forms also implies that $\Delta = \beta_1{:}\kappa_1, \ldots, \beta_n{:}\kappa_1$ and therefore $[c_1, \ldots, c_n/\Delta]$ is a substitution. Therefore,

$$(S, \mathtt{coerce}(\gamma); e)v \longmapsto_A (S'S''\{\ell \mapsto h'\}, e[c_1, \ldots, c_n/\Delta])$$

Definition 37 (Stuck Program) *A program (S, e) is stuck if no operational rule applies and e is not* $\mathtt{halt}$ i.

Theorem 38 (Alias Type Soundness) *If $\vdash_A (S, e)$ and $(S, e) \longmapsto^*_A (S', e')$ then (S', e') is not stuck.*

Proof: By induction on the length of the evaluation sequence. Uses Progress and Preservation.

Theorem 39 (Alias Type Complete Collection)
If $\vdash_A (S, e)$ and $(S, e) \longmapsto^*_A (S', \mathtt{halt}\ v)$ then for some location ℓ, $v = \mathtt{ptr}(\ell)$ and $S' = \{\ell \mapsto \mathtt{union}_{\langle\mathcal{S}(1)\rangle\cup\langle\mathcal{S}(2)\rangle}(\langle\mathcal{S}(i)\rangle)\}$

Proof:

By Progress and Preservation and induction on the length of the evaluation sequence, $\vdash_A (S', \mathtt{halt}\ v)$. By inspection of the typing judgement for $\mathtt{halt}$ and Canonical Value Forms, $v = \mathtt{ptr}(\ell)$ for some location ℓ. By inspection of the typing judgement again and Canonical Store Forms, there exists an h such that

1. $S' = S''\{\ell \mapsto h\}$
2. $\vdash_A S'' : \emptyset$
3. $\vdash_A h : \langle\mathcal{S}(1)\rangle \cup \langle\mathcal{S}(2)\rangle$

From 3 and Canonical Value Forms, we can conclude that

$$h = \mathtt{union}_{\langle\mathcal{S}(1)\rangle\cup\langle\mathcal{S}(2)\rangle}(\langle\mathcal{S}(i)\rangle)$$

for some i. By induction on the type equality judgement, we can deduce that $\emptyset$ is equal to $\emptyset$ and no other store type. Therefore, from 2 and inspection of the store typing rule, we know that S'' must be the empty map and consequently, from 1, S' is simply equal to $\{\ell \mapsto \mathtt{union}_{\langle\mathcal{S}(1)\rangle\cup\langle\mathcal{S}(2)\rangle}(\langle\mathcal{S}(i)\rangle)\}$.

□

Appendix B

Capability Calculus Type Soundness & Garbage Collection Properties

Overview The proof is broken down into a series of lemmas, most of which are proven by induction on the typing derivations or by induction on the syntax of the language. The proof culminates in a proof Type Soundness and Complete Collection. The supporting lemmas are grouped as follows:

- Lemmas 40 to 42 describe when extensions to type contexts or exchanges of elements within a type context are permissible.
- Lemmas 43 to 45 state that constructors involved in equality and subtyping judgements are well-formed and that all free variables of well-formed constructors are bound by the type context.
- Definitions and lemmas 46 to 53 describe which capabilities are equal to one another and which capabilities are subtypes of one another. They provide a higher level of abstraction than the rules for equality and subtyping and are used frequently in the rest of the proof.
- Lemmas 54 and 55 are substitution lemmas for types and values respectively.
- Lemma 56 states that well-formed small values, heap values, and declarations have well-formed types.
- Lemmas 57 to 59 are Canonical Forms lemmas. Given a type, these lemmas describe the shape of memory or of values.
- Lemmas 60 to 62 describe the conditions under which you can add labels or regions to the memory type and preserve typing.
- Lemma 64 states that satisfiability is preserved across equality (under the empty context).
- Lemma 65 states that satisfiability is preserved when a region and the corresponding unique capability are removed both from the store and the current capability simulataneously.
- Lemmas 66 and 67 are the Preservation and Progress lemmas respectively. They are used directly in the proof of Type Soundness.

Lemma 40 *If* $\Delta \vdash_R \Delta'$ *then* $Dom(\Delta) \cap Dom(\Delta') = \emptyset$.

Proof:

By induction on the derivation.

□

Lemma 41 (Type Context Exchange) *If* $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$ *then*

1. *If* $\Delta_0\Delta_1\Delta_2\Delta_3 \vdash_R \Delta$ *then* $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash_R \Delta$
2. *If* $\Delta_0\Delta_1\Delta_2\Delta_3 \vdash_R c : \kappa$ *then* $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash_R c : \kappa$

Proof:

By induction on the derivations. In the rule type-var:

$$\frac{}{\Delta_0\Delta_1\Delta_2\Delta_3 \vdash_R \alpha : \kappa} \; (\Delta_0\Delta_1\Delta_2\Delta_3(\alpha) = \kappa)$$

we know $\Delta_0\Delta_1\Delta_2\Delta_3(\alpha) = \Delta_0\Delta_2\Delta_1\Delta_3(\alpha)$ because the domains of Δ_1 and Δ_2 are disjoint. Consequently, $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash_R \alpha : \kappa$.

□

Lemma 42 (Type Context Extension) *If* $\Delta \vdash_R \Delta'$ *then*

1. *If* $\Delta \vdash_R \Delta''$ *and* $Dom(\Delta'') \cap Dom(\Delta') = \emptyset$ *then* $\Delta\Delta' \vdash_R \Delta''$
2. *If* $\Delta \vdash_R c : \kappa$ *then* $\Delta\Delta' \vdash_R c : \kappa$
3. *If* $\Delta \vdash_R c_1 = c_2 : \kappa$ *then* $\Delta\Delta' \vdash_R c_1 = c_2 : \kappa$
4. *If* $\Delta \vdash_R c_1 \leq c_2 : \kappa$ *then* $\Delta\Delta' \vdash_R c_1 \leq c_2 : \kappa$

Proof:

By induction on the derivation. Almost all cases follow directly from the inductive hypothesis. Rules ctxt-sub and type-arrow require Type Context Exchange where Δ_3 is $\cdot$.

□

Lemma 43 *If* $\Delta \vdash_R c : \kappa$ *then* $\mathrm{FV}(c) \subseteq Dom(\Delta)$.

Proof: By induction on the derivation.

Lemma 44 (Equality Regularity) *If* $\Delta \vdash_R C = C' : \kappa$ *then* $\Delta \vdash_R C : \kappa$ *and* $\Delta \vdash_R C' : \kappa$.

Proof: By induction on the derivation.

Lemma 45 (Subtyping Regularity) *If* $\cdot \vdash_R \Delta$ *and* $\Delta \vdash_R C \leq C' : \kappa$ *then* $\Delta \vdash_R C : \kappa$ *and* $\Delta \vdash_R C' : \kappa$.

Proof:

By induction on the derivation. In the rule sub-var, we show by induction on the derivation $\cdot \vdash_R \Delta$, that if $(\epsilon \leq C) \in \Delta$ then $\Delta \vdash_R \Delta(\epsilon) : \texttt{Store}$.

□

Definition 46 (Atomic Element) *An atomic element, a, is a type variable ϵ of kind* `Store`*, a singleton capability $\{r\}$, or a barred capability $\overline{\epsilon}$ or $\overline{\{r\}}$. The meta-variable a ranges over atomic elements.*

Definition 47 *$E(C)$ is the set of elements ϵ or $\{r\}$ that appear in C (where $\{\!|x_1, \ldots, x_n|\!\}$ is notation for the* set *of elements $x_1, \ldots, x_n$):*

$$\begin{array}{rcl} E(\emptyset) & = & \emptyset \\ E(\{r\}) & = & \{\!|\{r\}|\!\} \\ E(\epsilon) & = & \{\!|\epsilon|\!\} \\ E(C_1 * C_2) & = & E(C_1) \cup E(C_2) \\ E(\overline{C}) & = & E(C) \end{array}$$

Lemma 48 (Equality) *If $\Delta \vdash_R C : \texttt{Store}$ then*

1. $\Delta \vdash_R C = a_1 * \cdots * a_n : \texttt{Store}$ *for some atomic capabilities $a_1, \ldots, a_n$*
2. $\Delta \vdash_R a_1 * \cdots * a_{i-1} * a_i * a_{i+1} * \cdots * a_n = a_1 * \cdots * a_{i-1} * a_{i+1} * \cdots * a_n * a_i : \texttt{Store}$
3. $\Delta \vdash_R a_1 * \cdots * a_n = a'_1 * \cdots * a'_n : \texttt{Store}$ *where $a'_1, \ldots, a'_n$ is any permutation of $a_1, \ldots, a_n$.*
4. $\Delta \vdash_R a_1 * \cdots * a_n = a'_1 * \cdots * a'_m : \texttt{Store}$ *where $a'_1, \ldots, a'_m$ is a subsequence of $a_1, \ldots, a_n$ with all duplicate barred elements removed.*
5. *If $\Delta \vdash_R C = C' : \texttt{Store}$ then the sets $E(C)$ and $E(C')$ are equal.*
6. *If $E(C) = E(C')$ and $\Delta \vdash_R C' : \texttt{Store}$ then $\Delta \vdash_R \overline{C} = \overline{C'} : \texttt{Store}$.*
7. *If $\Delta \vdash_R C * \{r\} = C' * \{r\} : \texttt{Store}$ then $\Delta \vdash_R C = C' : \texttt{Store}$.*

Proof:

Part 1 follows by induction on the derivation $\Delta \vdash_R C : \texttt{Store}$. Case type-$\emptyset$ is immediate. Case type-single, follows from application of the equality rules R-eq-symm and R-eq-$\emptyset$. Case type-plus is more intricate. The inductive hypothesis gives us:

$$\begin{array}{c} \Delta \vdash_R C_1 = a_1 * \cdots * a_n : \texttt{Store} \\ \Delta \vdash_R C_2 = a'_1 * \cdots * a'_m : \texttt{Store} \end{array}$$

By induction on m and using the rules R-eq-$\emptyset$, R-eq-assoc, and R-eq-trans

$$\Delta \vdash_R a'_1 * \cdots * a'_m = a'_1 * (a'_2 * \cdots * (a'_{m-1} * a'_m) \cdots) : \texttt{Store}$$

By equality congruence and R-eq-trans,

$$\Delta \vdash_R C_1 * C_2 = (a_1 * \cdots * a_n) * a'_1 * (a'_2 * \cdots * (a'_{m-1} * a'_m) \cdots) : \texttt{Store}$$

By induction on m again and using R-eq-assoc, R-eq-symm, and R-eq-trans,

$$\Delta \vdash_R C_1 * C_2 = a_1 * \cdots * a_n * a'_1 * \cdots * a'_m : \texttt{Store}$$

For the case $\overline{C}$, we have $\Delta \vdash_R C = a_1 * \cdots * a_n : \texttt{Store}$ by IH. By congruence, $\Delta \vdash_R \overline{C} = \overline{a_1 * \cdots * a_n} : \texttt{Store}$. By induction on n, $\Delta \vdash_R \overline{C} = \overline{a_1} * \cdots * \overline{a_n} : \texttt{Store}$. For each a_i, either $\overline{a_i}$

is an atomic element or a_i is already barred and we use the R-eq-bar-idem rule to show that $\Delta \vdash_R \overline{a_i} = a_i : \texttt{Store}$. In either case, by induction on n again and use of the congruence rules, we are done.

Part 2 follows by induction on $m - i$ using R-eq-assoc, R-eq-comm as well as the transitivity and symmetry of equality. Part 3 is a corollary of part 2. Part 4 follows by induction on the number of barred duplicates and uses part 3, transitivity, symmetry, and R-eq-dup rules. Part 5 follows by induction on the equality judgement.

Part 6 may be proven as follows:
$\Delta \vdash_R C = a_1 * \cdots * a_n : \texttt{Store}$ where $E(C) = E(a_1 * \cdots * a_n)$ by parts 1 and 5.
$\Delta \vdash_R C' = a'_1 * \cdots * a'_m : \texttt{Store}$ where $E(C') = E(a'_1 * \cdots * a'_m)$ by parts 1 and 5.
By parts 3 and 4 and congruence of equality: $\Delta \vdash_R \overline{C} = \overline{a_1 * \cdots * a_n} = a_{j_1} * a_{j_n} : \texttt{Store}$
$\Delta \vdash_R \overline{C'} = \overline{a'_1 * \cdots * a'_m} = a'_{j_1} * a'_{j_m} : \texttt{Store}$
where the a_{j_i} and a'_{j_i} contain no duplicates and are ordered according to some canonical ordering. If $E(C) = E(C')$ then the a_{j_i} and the a'_{j_i} are the same and are in the same order. Hence, the constructors are syntactically equal and thus definitionally equal.

Part 7 follows by induction on the typing derivation.

□

Definition 49 (Linear/Non-Linear Capabilities) *A capability C is* linear *in C' if there does not exist C'' such that $\overline{C''} \in C'$ and $C \in C''$. A capability C is* non-linear *in C' if $\overline{C''} \in C'$ and $C \in \overline{C''}$.*

Lemma 50 *If $\Delta \vdash_R C' : \texttt{Store}$ and C is non-linear in C' then $\Delta \vdash_R C' = C'' * \overline{C} : \texttt{Store}$.*

Proof: By induction on the typing derivation.

Lemma 51 *If $\Delta \vdash_R C' : \texttt{Store}$ and C is linear in C' then $\Delta \vdash_R C' = C'' * C : \texttt{Store}$.*

Proof: By induction on the typing derivation.

Lemma 52 (Capability Equality Cardinality Preservation(CECP))
If $\Delta \vdash_R C_1 = C_2 : \texttt{Store}$ and $\Delta \vdash_R a : \texttt{Store}$ and $a = \epsilon$ or $\{r\}$ then

1. *a is linear (non-linear) in C_1 iff a is linear (non-linear) in C_2.*
2. *The number of linear occurences of a is the same in C_1 and C_2.*

Proof: By induction on the derivation.

Lemma 53 (Capability Subtyping Cardinality Preservation(CSCP))
If $\cdot \vdash_R C_1 \leq C_2 : \texttt{Store}$ then

1. *For all region names ν, $\{\nu\} \in C_1$ iff $\{\nu\} \in C_2$.*
2. *For all region names ν, if $\{\nu\}$ is not linear in C_1 then $\{\nu\}$ is not linear in C_2.*

Proof:

By induction on the derivation and Capability Equality Cardinality Preservation. Note that by Subtyping Regularity and Lemma 43, no type variables ϵ appear in C_1 and consequently, the rule sub-var never appears in the derivation.

□

Lemma 54 (Type Substitution)
Let
Δ_0 *be* $\Delta[c_1, \ldots, c_n/\Delta']$ Γ_0 *be* $\Gamma[c_1, \ldots, c_n/\Delta']$ C_0 *be* $C[c_1, \ldots, c_n/\Delta']$
r_0 *be* $r[c_1, \ldots, c_n/\Delta']$ τ_0 *be* $\tau[c_1, \ldots, c_n/\Delta']$.
If Δ' *is* $b_1, \ldots, b_n$ *where for* $1 \leq i \leq n$*:*

A1. if b_i *is* $\alpha_i{:}\kappa_i$ *then* $\cdot \vdash_R c_i : \kappa_i$

A2. if b_i *is* $\epsilon_i \leq C_i$ *then* $\cdot \vdash_R c_i \leq C_i : \mathtt{Store}$

then

1. *If* $\Delta', \Delta \vdash_R \Delta''$ *then* $\Delta_0 \vdash_R \Delta''[c_1, \ldots, c_n/\Delta']$
2. *If* $\Delta', \Delta \vdash_R c : \kappa$ *then* $\Delta_0 \vdash_R c[c_1, \ldots, c_n/\Delta'] : \kappa$
3. *If* $\Delta', \Delta \vdash_R c = c' : \kappa$ *then* $\Delta_0 \vdash_R c[c_1, \ldots, c_n/\Delta'] = c'[c_1, \ldots, c_n/\Delta'] : \kappa$
4. *If* $\Delta', \Delta \vdash_R c \leq c' : \kappa$ *then* $\Delta_0 \vdash_R c[c_1, \ldots, c_n/\Delta'] \leq c'[c_1, \ldots, c_n/\Delta'] : \kappa$
5. *If* $\Delta', \Delta \vdash_R \Delta_1 = \Delta_2$ *then* $\Delta_0 \vdash_R \Delta_1[c_1, \ldots, c_n/\Delta'] = \Delta_2[c_1, \ldots, c_n/\Delta']$
6. *If* $\Psi; \Delta', \Delta; \Gamma; r \vdash_R h : \tau$ *then* $\Psi; \Delta_0; \Gamma_0; r_0 \vdash_R h[c_1, \ldots, c_n/\Delta'] : \tau_0$
7. *If* $\Psi; \Delta', \Delta; \Gamma \vdash_R v : \tau$ *then* $\Psi; \Delta_0; \Gamma_0 \vdash_R v[c_1, \ldots, c_n/\Delta'] : \tau_0$
8. *If* $\Psi; \Delta', \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta', \Delta''; \Gamma''; C''$ *then*
   $\Psi; \Delta_0; \Gamma_0; C_0 \vdash_R d[c_1, \ldots, c_n/\Delta'] \Longrightarrow (\Delta''; \Gamma''; C'')[c_1, \ldots, c_n/\Delta']$.
9. *If* $\Psi; \Delta', \Delta; \Gamma; C \vdash_R e$ *then* $\Psi; \Delta_0; \Gamma_0; C_0 \vdash_R e[c_1, \ldots, c_n/\Delta']$

Proof:

By induction on the derivations. Almost all cases follow directly from the IH. In part 2, we must prove our lemma for the rules:

$$\frac{}{\Delta', \Delta \vdash_R \alpha : \kappa}\ (\Delta', \Delta(\alpha) = \kappa) \qquad \frac{}{\Delta', \Delta \vdash_R \epsilon : \mathtt{Store}}\ ((\epsilon \leq C) \in \Delta', \Delta)$$

In the first case, we have our result by A1 and Type Context Extension. In the second case, assume ϵ is ϵ_i. By A2, we have $\cdot \vdash_R c_i \leq C_i : \mathtt{Store}$. Because $\cdot \vdash_R \cdot$, Subtyping Regularity tells us that $\cdot \vdash_R c_i : \mathtt{Store}$. By Type Context Extension $\Delta_0 \vdash_R c_i : \mathtt{Store}$. In part 4, for the rule:

$$\frac{}{\Delta', \Delta \vdash_R \epsilon \leq C : \mathtt{Store}}\ ((\epsilon \leq C) \in \Delta', \Delta)$$

our result follows by A2 and Type Context Extension. In part 9, the case for `let`, we can apply the induction hypothesis because inspection of the rules for declarations show that $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta, \Delta''; \Gamma''; C'$ instead of the more general $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta''; \Gamma''; C'$.

□

Lemma 55 (Value Substitution) *If* Γ *is* $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$, $\cdot \vdash_R \Gamma$ *and for* $1 \leq i \leq n$, $\Psi; \cdot; \cdot \vdash_R v_i : \tau_i$ *then*

1. *If* $\Psi; \Delta; \Gamma, \Gamma' \vdash_R h \ \mathtt{at}\ r : \tau$ *then* $\Psi; \Delta; \Gamma' \vdash_R h[v_1, \ldots, v_n/x_1, \ldots, x_n]\ \mathtt{at}\ r : \tau$
2. *If* $\Psi; \Delta; \Gamma, \Gamma' \vdash_R v : \tau$ *then* $\Psi; \Delta; \Gamma' \vdash_R v[v_1, \ldots, v_n/x_1, \ldots, x_n] : \tau$
3. *If* $\Psi; \Delta; \Gamma, \Gamma'; C \vdash_R d \Longrightarrow \Delta'; \Gamma, \Gamma''; C'$
then $\Psi; \Delta; \Gamma'; C \vdash_R d[v_1, \ldots, v_n/x_1, \ldots, x_n] \Longrightarrow \Delta'; \Gamma''; C'$
4. *If* $\Psi; \Delta; \Gamma, \Gamma'; C \vdash_R e$ *then* $\Psi; \Delta; \Gamma'; C \vdash_R e[v_1, \ldots, v_n/x_1, \ldots, x_n]$

Proof:

By induction on the typing derivations. In part 4, the case for `let`, we can use the induction hypothesis because inspection of the typing rules for declarations reveals that $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma, \Gamma'; C'$ instead of the more general $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$.

□

Lemma 56 (Term Judgement Regularity)
If

A1 $\vdash_R \Psi$

A2 $\cdot \vdash_R C : \mathtt{Store}$

A3 $\cdot \vdash_R r : \mathtt{Rgn}$

then

1. *If* $\Psi; \cdot; \cdot \vdash_R v : \tau$ *then* $\cdot \vdash_R \tau : \mathtt{Type}$
2. *If* $\Psi; \cdot; \cdot \vdash_R h \ \mathtt{at}\ r : \tau$ *then* $\cdot \vdash_R \tau : \mathtt{Type}$
3. *If* $\Psi; \cdot; \cdot; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$ *then* $\cdot \vdash_R \Delta'$ *and* $\Delta' \vdash_R \Gamma'$ *and* $\Delta' \vdash_R C' : \mathtt{Store}$

Proof:

By induction on the typing derivations. Almost all cases follow directly from the induction hypothesis and Equality Regularity or Subtyping Regularity. In part 1, consider the case for type application:

$$\frac{\Psi; \cdot; \cdot \vdash_R v : \forall[\alpha{:}\kappa, \Delta].(C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ r \qquad \cdot \vdash_R c : \kappa}{\Psi; \cdot; \cdot \vdash_R v[c] : (\forall[\Delta].(C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0})[c/\alpha]\ \mathtt{at}\ r}$$

By the induction hypothesis, and then inspection of the typing rules for arrow types, we can deduce a judgement of the following form:

$$\frac{\dfrac{\alpha{:}\kappa \vdash_R \Delta}{\cdot \vdash_R \alpha{:}\kappa, \Delta} \quad \alpha{:}\kappa, \Delta \vdash_R C' : \mathtt{Store} \quad \alpha{:}\kappa, \Delta \vdash_R \tau_i : \mathtt{Type} \quad (\text{for } 1 \leq i \leq n)}{\cdot \vdash_R \forall[\alpha{:}\kappa, \Delta].(C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ r : \mathtt{Type}}$$

By Type Substitution, we may deduce that $\cdot \vdash_R (\forall[\Delta'].(C', \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0})[c/\alpha]\ \mathtt{at}\ r$. The second type application rule follows similarly.

□

Lemma 57 (Canonical Memory Forms)
If $\vdash_R \{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \{\nu_1{:}\Upsilon_1 \ldots, \nu_n{:}\Upsilon_n\}$ *then for all* $1 \leq i \leq n$ *and for all* $\ell \in Dom(\Upsilon_i)$*, either*

1. $\Upsilon_i(\ell)$ *is* $\langle\tau_1, \ldots, \tau_m\rangle\ \mathtt{at}\ \nu_i$ *and* $R_i(\ell) = \langle v_1, \ldots, v_m\rangle$ *and for* $1 \leq j \leq m$, $\Psi; \cdot; \cdot \vdash_R v_j : \tau_j$ *or*

2. $\Upsilon_i(\ell)$ *is* $\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ \nu_i$
and $R_i(\ell) = \mathtt{fix} f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
and $\Psi; \Delta; \{f{:}\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ \nu_i, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C \vdash_R e$

Proof: By inspection of the typing judgements for the store, regions, and heap values.

Lemma 58 (Canonical Memory Forms II)

1. *If* $\vdash_R S : \Psi$ *and* $\nu \notin S$ *then* $\nu \notin \Psi$

2. *If* $\Psi \vdash_R R\ \mathtt{at}\ \nu' : \Upsilon$ *and* $\nu \notin R$ *and* ν' *is not* ν *then* $\nu \notin \Upsilon$

3. *If* $\Psi; \cdot; \cdot \vdash_R h\ \mathtt{at}\ \nu' : \tau$ *and* ν' *is not* ν *and* $\nu \notin h$ *then* $\nu \notin \tau$

4. *If* $\Psi; \cdot; \cdot \vdash_R v : \tau$ *and* $\nu \notin v$ *then* $\nu \notin \tau$

Proof: By induction on the typing derivations.

Lemma 59 (Canonical Forms) *If* $\vdash_R S : \Psi$ *and* $\Psi; \cdot; \cdot \vdash_R v : \tau$ *then*

1. *If* τ *is int then* $v = i$.

2. *If* τ *is handle*(ν) *then* $v = \mathtt{handle}(\nu)$.

3. *If* τ *is* $\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \mathtt{at}\ \nu$ *then*

   (a) $v = \nu.\ell[c_1, \ldots, c_m]$
   (b) $S(\nu.\ell) = \mathtt{fix} f[\Delta', \Delta''](C', x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n).e$
   (c) Δ' *is* $b_1, \ldots, b_m$ *and for* $0 \leq i \leq m$, *either* b_i *is* $\alpha_i{:}\kappa_i$ *and* $\cdot \vdash_R c_i{:}\kappa_i$, *or* b_i *is* $\epsilon_i \leq C_i$ *and* $\cdot \vdash_R c_i \leq C_i : \mathtt{Store}$.
   (d) $\cdot \vdash_R \Delta = \Delta''[c_1, \ldots, c_m/\Delta']$, *and* $\Delta \vdash_R C = C'[c_1, \ldots, c_m/\Delta']$, *and for* $1 \leq i \leq n$, $\Delta \vdash_R \tau_i = \tau'_i[c_1, \ldots, c_m/\Delta'] : \mathtt{Type}$
   (e) $\Psi; \Delta', \Delta''; \{f{:}\forall[\Delta', \Delta''].(C', \tau'_1, \ldots, \tau'_n) \rightarrow \mathbf{0}\ \mathtt{at}\ \nu, x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n\}; C' \vdash_R e$

   or instead of (b),(c),(d), and (e): $\nu \notin \Psi$.

4. *If* τ *is* $\langle\tau_1, \ldots, \tau_n\rangle\ \mathtt{at}\ \nu$ *then*

   (a) $v = \nu.\ell$
   (b) $S(\nu.\ell) = \langle v_1, \ldots, v_n\rangle$
   (c) $\Psi; \cdot \vdash_R v_i : \tau_i$

or instead of (b),(c): $\nu \notin \Psi$.

Proof:

Part 1, 2 follow by inspection of the typing rules for word values.
Part 3 follows by induction on the derivation, $\Psi; \cdot; \cdot \vdash_R v : \tau$. By Canonical Memory Forms and inspection of the typing rules for word values, either $\nu.\ell$ or one of the type application rules are last:

case $\nu.\ell$ where $\nu \notin \Psi$:

(a) Trivial.

case $\nu.\ell$ where $\nu \in \Psi$:

(a) Trivial.
(b) By Canonical Memory Forms where Δ' is $\cdot$, Δ'' is Δ, C' is C, and for $1 \leq i \leq n$, τ_i' is τ_i.
(c) Trivial.
(d) Trivial.
(e) By inspection of judgement.

case $v[C]$

$$\frac{\Psi; \cdot; \cdot \vdash_R v : \forall[\epsilon \leq C_a, \Delta].(C_b, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\ \texttt{at}\ r \qquad \cdot \vdash_R C \leq C_a : \texttt{Store}}{\Psi; \cdot; \cdot \vdash_R v[C] : (\forall[\Delta].(C_b, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0})[C/\epsilon]\ \texttt{at}\ r}$$

By Term Judgement Regularity and Lemma 43, r is ν. The inductive hypothesis is as follows:

(a) $v = \nu.\ell[c_1, \ldots, c_n]$
(b) $S(\nu.\ell) = \texttt{fix} f[\Delta', \epsilon \leq C_a', \Delta''](C_b', x_1{:}\tau_1', \ldots, x_n{:}\tau_n').e$
(c) Δ' is $b_1, \ldots, b_m$ and for $0 \leq i \leq m$, either b_i is $\alpha_i{:}\kappa_i$ and $\cdot \vdash_R c_i{:}\kappa_i$, or b_i is $\epsilon_i \leq C_i$ and $\cdot \vdash_R c_i \leq C_i : \texttt{Store}$.
(d) $\cdot \vdash_R \epsilon \leq C_a, \Delta = (\epsilon \leq C_a', \Delta'')[c_1, \ldots, c_m/\Delta']$
and $\epsilon \leq C_a, \Delta \vdash_R C_b = C_b'[c_1, \ldots, c_m/\Delta']$
and for $1 \leq i \leq n$, $\epsilon \leq C_a, \Delta \vdash_R \tau_i = \tau_i'[c_1, \ldots, c_m] : \texttt{Type}$
(e) $\Psi; \Delta', \epsilon \leq C_a, \Delta''; \Gamma; C_b' \vdash_R e$
where $\Gamma = \{f{:}\forall[\Delta', \epsilon \leq C_a, \Delta''].(C_b', \tau_1', \ldots, \tau_n') \rightarrow \mathbf{0}\ \texttt{at}\ \nu, x_1{:}\tau_1', \ldots, x_n{:}\tau_n'\}$

or instead of (b),(c),(d), and (e), $\nu \notin \Psi$. Thus,

(a) $v[C] = \nu.\ell[c_1, \ldots, c_n, C]$ from IH.
If $\nu \notin \Psi$ then the result follows trivially. Thus assume $\nu \in \Psi$.
(b) By IH.
(c) By IH and the typing judgement which states $\cdot \vdash_R C \leq C_a : \texttt{Store}$.
(d) By IH and Type Substitution.
(e) By IH.

case $v[c]$ Similar.

Part 4 follows by inspection of the typing rules for word values. Notice only the $\nu.\ell$ rule when $\nu \in \Psi$, or the rule for tuples when $\nu \notin \Psi$ can apply. Assuming the former (the latter is trivial), then (a) is immediate and (b), (c) follow by Canonical Memory Forms.

□

Lemma 60 (Memory Type GC) *If* $\vdash_R \Psi$ *and* Ψ' *is* $\Psi \backslash \nu$ *then*

1. *If* $\Psi; \Delta; \Gamma \vdash_R h \; \mathtt{at} \; r : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R h \; \mathtt{at} \; r : \tau$
2. *If* $\Psi; \Delta; \Gamma \vdash_R v : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R v : \tau$
3. *If* $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$
4. *If* $\Psi; \Delta; \Gamma; C \vdash_R e$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R e$
5. *If* $\Psi \vdash_R R \; \mathtt{at} \; \nu : \Upsilon$ *then* $\Psi' \vdash_R R \; \mathtt{at} \; \nu : \Upsilon$

Proof:

By induction on the typing derivation. All cases follow directly from IH except the rule:

$$\frac{}{\Psi; \Delta; \Gamma \vdash_R \nu'.\ell : \tau} \; (\Psi(\nu'.\ell) = \tau)$$

When ν is not ν', this case is trivial so assume ν is ν'. By Canonical Memory Forms, τ is either $\forall[\Delta'].(\epsilon, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0} \; \mathtt{at} \; \nu$ or $\langle \tau_1, \ldots, \tau_n \rangle \; \mathtt{at} \; \nu$. Because $\vdash_R \Psi$, we have $\cdot \vdash_R \tau : \mathtt{Type}$. By Type Context Extension, $\Delta \vdash_R \tau : \mathtt{Type}$. Thus, in either of the above cases, $\Psi; \Delta; \Gamma \vdash_R \nu'.\ell : \tau$ via one of the two rules for $\nu \notin Dom(\Psi)$.

□

Lemma 61 (Memory Type Extension) *If* ν *does not appear in* Ψ, Δ, Γ, r, h, v, d, e, *or* R, *and* Ψ' *is* $\Psi\{\nu{:}\{\,\}\}$ *then*

1. *If* $\Psi; \Delta; \Gamma \vdash_R h \; \mathtt{at} \; r : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R h \; \mathtt{at} \; r : \tau$
2. *If* $\Psi; \Delta; \Gamma \vdash_R v : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R v : \tau$
3. *If* $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$
4. *If* $\Psi; \Delta; \Gamma; C \vdash_R e$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R e$
5. *If* $\Psi \vdash_R R \; \mathtt{at} \; \nu' : \Upsilon$ *then* $\Psi' \vdash_R R \; \mathtt{at} \; \nu' : \Upsilon$

Proof:

By induction on the typing derivation. In part 2, for the rule:

$$\frac{\Delta \vdash_R \langle \tau_1, \ldots, \tau_n \rangle \; \mathtt{at} \; \nu' : \mathtt{Type}}{\Psi; \Delta; \Gamma \vdash_R \nu'.\ell : \langle \tau_1, \ldots, \tau_n \rangle \; \mathtt{at} \; \nu'} \; (\nu' \notin \Psi)$$

ν' is not ν by assumption and thus the result holds and similarly for the analogous rule for arrow types.

□

Lemma 62 (Region Type Extension) *If* $\nu \in Dom(\Psi)$, $\ell \notin Dom(\Psi(\nu))$, *and* Ψ' *is* $\Psi\{\nu.\ell{:}\tau\}$ *then*

1. *If* $\Psi; \Delta; \Gamma \vdash_R h \texttt{ at } r : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R h \texttt{ at } r : \tau$

2. *If* $\Psi; \Delta; \Gamma \vdash_R v : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash_R v : \tau$

3. *If* $\Psi; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R d \Longrightarrow \Delta'; \Gamma'; C'$

4. *If* $\Psi; \Delta; \Gamma; C \vdash_R e$ *then* $\Psi'; \Delta; \Gamma; C \vdash_R e$

5. *If* $\Psi \vdash_R R \texttt{ at } \nu' : \Upsilon$ *then* $\Psi' \vdash_R R \texttt{ at } \nu' : \Upsilon$

6. *If* $\Psi \vdash_R C \text{ sat}$ *then* $\Psi' \vdash_R C \text{ sat}$

Proof:

By induction on the typing derivation.

□

Lemma 63 *If* $\Delta \vdash_R C_1 * C_2 : \texttt{Store}$ *and* $\Delta \vdash_R C_1 * C_2 = a_1 * \cdots * a_n : \texttt{Store}$ *then* $\Delta \vdash_R C_1 = a'_1 * \cdots * a'_m : \texttt{Store}$ *and* $a'_1, \ldots, a'_m$ *is a subset of* $a_1, \ldots, a_n$.

Proof:

By Lemma 48 (1), $\Delta \vdash_R C_1 = a'_1 * \cdots * a'_m : \texttt{Store}$. By Lemma 48 (5), $E(a'_1 * \cdots * a'_m) = E(C_1) \subseteq E(C_1 * C_2) = E(a_1 * \cdots * a_n)$. By CECP, a_i is linear (non-linear) in $a'_1 * \cdots * a'_m$ if and only if a_i is linear (non-linear) in $a_1 * \cdots * a_n$. Therefore, $a \in a'_1, \ldots, a'_m$ implies $a \in a_1, \ldots, a_n$.

□

Lemma 64 (Capability Satisfiability Preservation) *If* $\Psi \vdash_R C \text{ sat}$ *and* $\cdot \vdash_R C = C' : \texttt{Store}$ *then* $\Psi \vdash_R C' \text{ sat}$.

Proof:

By symmetry and transitivity of equality and inspection of the sat derivation.

□

Lemma 65 *If* $\Psi \vdash_R C * \{\nu\} \text{ sat}$ *then* $\Psi \backslash \nu \vdash_R C \text{ sat}$.

Proof:

1. Assume $\Psi \vdash_R C * \{\nu\} \text{ sat}$
2. and $\Psi = \{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}$
3. From 1, we know $\cdot \vdash_R C * \{\nu\} = \{\nu_1, \ldots, \nu_n\} : \texttt{Store}$
4. and $\nu_i \neq \nu_j$, for $1 \leq i, j \leq n$ and $i \neq j$
5. From 3, 4, and CECP, ν appears once in $\{\nu_1, \ldots, \nu_n\}$ and once in $C * \{\nu\}$.
6. From 5, and Equality (3),
$\cdot \vdash_R \{\nu_1, \ldots, \nu_{i-1}, \nu, \nu_{i+1}, \ldots, \nu_n\} = \{\nu_1, \ldots, \nu_{i-1}, \nu_{i+1}, \ldots, \nu_n\} * \{\nu\} : \texttt{Store}$
7. From 3, 6, transitivity of equality, and Equality (7),
$\cdot \vdash_R C = \{\nu_1, \ldots, \nu_{i-1}, \nu_{i+1}, \ldots, \nu_n\} : \texttt{Store}$
8. Hence, from 2,4,7 we have $\Psi \backslash \nu \vdash_R C \text{ sat}$

□

Lemma 66 (Preservation) *If* $\vdash_R (S,e)$ *and* $(S,e) \longmapsto_R (S',e')$ *then* $\vdash_R (S',e')$

Proof:

The proof proceeds by cases on the structure of e. In each case, we show the form of the typing judgement that can be inferred by inspection of the typing rules and refer to it throughout the case as "the typing judgement". Then we give the transition specified by the operational semantics. Using these two facts, we derive the result $\vdash_R (S',e')$.

- $\texttt{let}\, v$

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad (A)}{\vdash_R (S, \texttt{let}\, x = v\, \texttt{in}\, e)}$$

$$\frac{\dfrac{\Psi; \cdot; \cdot \vdash_R v : \tau}{\Psi; \cdot; \cdot; C \vdash_R x = v \Longrightarrow \cdot; \{x{:}\tau\}; C} \quad \Psi; \cdot; \{x{:}\tau\}; C \vdash_R e}{\Psi; \cdot; \cdot; C \vdash_R \texttt{let}\, x = v\, \texttt{in}\, e}\ (A)$$

and $(S, \texttt{let}\, x = v\, \texttt{in}\, e) \longmapsto_R (S, e[v/x])$. By the typing judgement and Value Substitution, $\vdash_R (S, e[v/x])$.

- $\texttt{let}\, h$

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad (A)}{\vdash_R (S, \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e)}$$

$$\frac{\dfrac{\dfrac{\Psi; \cdot; \cdot \vdash_R v : \mathit{handle}(\nu)}{\Psi; \cdot; \cdot \vdash_R h\, \texttt{at}\, \nu : \tau} \quad \cdot \vdash_R C \leq C' * \overline{\{\nu\}}}{\Psi; \cdot; \cdot; C \vdash_R x = h\, \texttt{at}\, v \Longrightarrow \cdot; \{x{:}\tau\}; C} \quad \dfrac{\vdots}{\Psi; \cdot; \{x{:}\tau\}; C \vdash_R e}}{\Psi; \cdot; \cdot; C \vdash_R \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e}\ (A)$$

where $v = \texttt{handle}(\nu)$
and $\nu \in \mathit{Dom}(S)$ and $\ell \notin \mathit{Dom}(S(\nu))$
and $S' = S\{\nu.\ell \mapsto h\}$
and $(S, \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e) \longmapsto_R (S', e[\nu.\ell/x])$
and let $\Psi' = \Psi\{\nu.\ell{:}\tau\}$

1. (a) τ is $\langle \tau_1, \ldots, \tau_n \rangle\, \texttt{at}\, \nu$ or $\forall[].(C'', \tau_1, \ldots, \tau_n) \to \mathbf{0}\, \texttt{at}\, \nu$ by inspection of the heap value typing rules and the typing judgement.
   (b) $\vdash_R S' : \Psi'$ by the typing judgement and inspection of the memory typing rule.
2. $\Psi' \vdash_R C$ sat by Region Type Extension.
3. (a) $\Psi'; \cdot; \cdot \vdash_R \nu.\ell : \tau$ by the typing rules for word values.
   (b) $\Psi'; \cdot; \cdot; C \vdash_R e[\nu.\ell/x]$ by (a) and Value Substitution.

By 1(b), 2, and 3(b), we have $\vdash_R (S', e[\nu.\ell/x])$.

- $\pi_i v$

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad (A)}{\vdash_R (S, \texttt{let}\, x = \pi_i v\, \texttt{in}\, e)}$$

$$\dfrac{\dfrac{\begin{array}{c}\Psi;\cdot;\cdot\vdash_R v:\langle\tau_1,\ldots,\tau_n\rangle\ \mathtt{at}\ \nu\\ \cdot\vdash_R C\le C'*\{\nu\}\end{array}}{\Psi;\cdot;\cdot;C\vdash_R x=\pi_i v\Longrightarrow\cdot;\{x{:}\tau_i\};C}\quad \Psi;\cdot;\{x{:}\tau_i\};C\vdash_R e}{\Psi;\cdot;\cdot;C\vdash_R \mathtt{let}\,x=\pi_i v\,\mathtt{in}\,e}\ (A)$$

where $v=\nu.\ell$
and $S(\nu.\ell)=\langle v_1,\ldots,v_n\rangle$
and $(S,\mathtt{let}\,x=\pi_i v\,\mathtt{in}\,e)\longmapsto_R(S,e[v_i/x])$

1. $\vdash_R S:\Psi$ by the typing judgement.
2. $\Psi\vdash_R C$ sat by the typing judgement.
3. (a) $\Psi;\cdot;\cdot\vdash_R v_i:\tau_i$ by Canonical Forms and the typing judgement.
   (b) $\Psi;\cdot;\cdot;C\vdash_R e[v_i/x]$ by Value Substitution, (a), and the typing judgement.

By 1,2,and 3(b), $\vdash_R(S,e[v_i/x])$.

- `freergn`

$$\dfrac{\vdash_R S:\Psi\quad \Psi\vdash_R C\ \text{sat}\quad (A)}{\vdash_R(S,\mathtt{freergn}\,v\,\mathtt{in}\,e)}$$

$$\dfrac{\dfrac{\Psi;\cdot;\cdot\vdash_R v:\mathit{handle}(\nu)\quad \cdot\vdash_R C\le C'*\{\nu\}}{\Psi;\cdot;\cdot;C\vdash_R\mathtt{freergn}\,v\Longrightarrow\cdot;\cdot;C'}\qquad \Psi;\cdot;\cdot;C'\vdash_R e}{\Psi;\cdot;\cdot;C\vdash_R\mathtt{freergn}\,v\,\mathtt{in}\,e}\ (A)$$

where v is $\mathtt{handle}(\nu)$ and $(S,\mathtt{freergn}\,v\,\mathtt{in}\,e)\longmapsto_R(S\backslash\nu,e)$. Let Ψ' be $\Psi\backslash\nu$.

1. $\vdash_R S':\Psi'$ by Memory Type GC and the typing judgement.
2. (a) $\Psi\vdash_R C$ sat and $\cdot\vdash_R C\le C'*\{\nu\}$ by the typing judgement.
   (b) $\Psi\vdash_R C'*\{\nu\}$ sat by Capability Satisfiability Preservation and (a)
   (c) $\Psi'\vdash_R C'$ sat by Lemma 65 and (b)
3. $\Psi';\cdot;\cdot;C'\vdash_R e$ by the typing judgement and Memory Type GC.

By 1, 2(e), and 3, $\vdash_R(S\backslash\nu,e)$.

- `newrgn`

$$\dfrac{\vdash_R S:\Psi\qquad \dfrac{\cdot\vdash_R C=C':\mathtt{Store}}{\Psi\vdash_R C\ \text{sat}}\ (\ldots)\qquad \dfrac{(A)\quad (B)}{\Psi;\cdot;\cdot;C\vdash_R\mathtt{newrgn}\ \rho,x_\rho\ \mathtt{in}\,e}}{\vdash_R(S,\mathtt{newrgn}\ \rho,x_\rho\ \mathtt{in}\,e)}$$

$$\dfrac{}{\Psi;\cdot;\cdot;C\vdash_R\mathtt{newrgn}\ \rho,x_\rho\Longrightarrow\rho{:}\mathtt{Rgn};\{x_\rho{:}\mathit{handle}(\rho)\};C*\{\rho\}}\ (A)$$

$$\dfrac{\vdots}{\Psi;\rho{:}\mathtt{Rgn};\{x_\rho{:}\mathit{handle}(\rho)\};C*\{\rho\}\vdash_R e}\ (B)$$

The operational rule is

$$(S,\mathtt{newrgn}\ \rho,x_\rho\ \mathtt{in}\,e)\longmapsto_R(S',e[\nu,\mathtt{handle}(\nu)/\rho,x_\rho])$$

where $S'=S\{\nu\mapsto\{\,\}\}$ and $\nu\notin S$ and $\nu\notin e$.
In what follows, let $\Psi'=\Psi\{\nu{:}\{\,\}\}$.

1. $\vdash_R \Psi'$ by Memory Type Extension and the typing judgement.
2. Since $\nu \notin \Psi$ by assumption in the operational semantics, we can satisfy the side condition on the sat judgement. We can also prove $\cdot \vdash_R C * \{\nu\} = C' * \{\nu\} : \texttt{Store}$ by the congruence rule for equality. Consequently, $\Psi' \vdash_R C * \{\nu\}$ sat.
3. $\Psi'; \cdot; \cdot; C * \{\nu\} \vdash_R e[\nu, \texttt{handle}(\nu)/\rho, x_\rho]$ from the typing judgement and application of Type and Value Substitution and then Memory Type Extension Lemmas.

By 1, 2(e), and 3, $\vdash_R (S', e[\nu, \texttt{handle}(\nu)/\rho, x_\rho])$.

- `if0`

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\Psi; \cdot; \cdot \vdash_R i : \mathit{int} \quad \Psi; \cdot; \cdot; C \vdash_R e_2 \quad \Psi; \cdot; \cdot; C \vdash_R e_3}{\Psi; \cdot; \cdot; C \vdash_R \texttt{if}\ i\ (e_2 \mid e_3)}}{\vdash_R (S, \texttt{if}\ i\ (e_2 \mid e_3))}$$

$(S, e) \longmapsto_R (S, e_2)$ if $i = 0$ and $(S, e) \longmapsto_R (S, e_3)$ otherwise. By the typing judgement, $\vdash_R (S, e_2)$, or $\vdash_R (S, e_3)$.

- $v_0(v_1, \ldots, v_n)$

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\begin{array}{c}\Psi; \cdot; \cdot \vdash_R v_i : \tau_i \quad (\text{for } 0 \le i \le n) \\ \cdot \vdash_R C \le C'' * \overline{\{\nu\}} : \texttt{Store} \\ \cdot \vdash_R \tau_0 = \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \texttt{at}\ \nu \\ \cdot \vdash_R C = C' : \texttt{Store}\end{array}}{\Psi; \cdot; \cdot; C \vdash_R v_0(v_1, \ldots, v_n)}}{\vdash_R (S, v_0(v_1, \ldots, v_n))}$$

$(S, v_0(v_1, \ldots, v_n)) \longmapsto_R (S, S(e))$
where $v_0 = \nu.\ell[c_1, \ldots, c_m]$
and $S(\nu.\ell) = \texttt{fix} f[b_1, \ldots, b_m](C'', x_1 : \tau_1, \ldots, x_n{:}\tau_n).e$
and for $1 \le i \le m$, $b_i = \alpha_i{:}\kappa_i$ or $b_i = \alpha_i \le C_i$
and $S = [c_1, \ldots, c_m, \nu.\ell, v_1, \ldots, v_n/\alpha_1, \ldots, \alpha_m, f, x_1, \ldots, x_n]$

1. $\vdash_R S : \Psi$ by the typing judgement.
2. $\Psi \vdash_R C'$ sat by Capability Satisfiability Preservation.
3. (a) $\Psi \vdash_R C'' * \overline{\{\nu\}}$ sat by Capability Satisfiability Preservation and the typing judgement
   (b) $\nu \in \mathit{Dom}(\Psi)$ by CECP and (a).
   (c) $\Psi; \cdot; \cdot \vdash_R v_0 : \forall[\,].(C''', \tau_1''', \ldots, \tau_n''') \to \mathbf{0}$ and $\cdot \vdash_R \forall[\,].(C''', \tau_1''', \ldots, \tau_n''') \to \mathbf{0} = \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0}$ by the typing judgement
   (d) $\Psi; b_1, \ldots, b_m; \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C'' \vdash_R e$ by Canonical Forms 3(e), (b), and (c).
   (e) $\cdot \vdash_R C' = C''[c_1, \ldots, c_m/\alpha_1, \ldots, \alpha_m] : \texttt{Store}$ by the transitivity of equality, Canonical Forms 3(d), (b), and (c)
   (f) $\Psi; \cdot; \cdot; C' \vdash_R S(e)$ by Type and Value Substitution, and (e).

By 1, 2, and 3(f), $\vdash_R (S, S(e))$

□

Lemma 67 (Progress) *If* $\vdash_R (S, e)$ *then either:*

1. *There exists* (S', e') *such that* $(S, e) \longmapsto_R (S', e')$*, or*

2. $e = \mathtt{halt}\, v$ *and* $\Psi; \cdot; \cdot \vdash_R v : int$.

Proof:

The proof proceeds by cases on the structure of e and makes heavy use of the Canonical Forms lemma.

- $\mathtt{let}\, v$ Trivial.
- $\mathtt{let}\, h$

$$\dfrac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\dfrac{\begin{array}{c}\Psi; \cdot; \cdot \vdash_R v : handle(r) \\ \Psi; \cdot; \cdot \vdash_R h\ \mathtt{at}\ r : \tau \\ \cdot \vdash_R C \leq C' * \overline{\{r\}}\end{array}}{\Psi; \cdot; \cdot; C \vdash_R x = h\ \mathtt{at}\ v \Longrightarrow \cdot; \{x{:}\tau\}; C} \quad \cdots}{\Psi; \cdot; \cdot; C \vdash_R \mathtt{let}\, x = h\ \mathtt{at}\ v\ \mathtt{in}\, e}}{\vdash_R (S, \mathtt{let}\, x = h\ \mathtt{at}\ v\ \mathtt{in}\, e)}$$

  $\Psi; \cdot; \cdot \vdash_R v : handle(r)$ directly from the typing judgement. By Term Judgement Regularity and Lemma 43, r is ν, and by Canonical Forms, v is the value $\mathtt{handle}(\nu)$. By Capability Satisfiability Preservation, $\Psi \vdash_R C' * \overline{\{\nu\}}$ sat and thus $\nu \in Dom(\Psi)$. By inspection of the memory typing rules, $\nu \in Dom(S)$. Thus $(S, e) \longmapsto_R (S\{\nu.\ell \mapsto h\}, e[\nu.\ell/x])$.

- $\pi_i v$

$$\dfrac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad (A)}{\vdash_R (S, \mathtt{let}\, x = \pi_i v\ \mathtt{in}\, e)}$$

$$\dfrac{\dfrac{\Psi; \cdot; \cdot \vdash_R v : \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ r \quad \cdot \vdash_R C \leq C' * \overline{\{r\}}}{\Psi; \cdot; \cdot; C \vdash_R x = \pi_i v \Longrightarrow \cdot; \{x{:}\tau_i\}; C} \quad \cdots}{\Psi; \cdot; \cdot; C \vdash_R \mathtt{let}\, x = \pi_i v\ \mathtt{in}\, e}\ (A)$$

  By Capability Satisfiability Preservation, $\Psi \vdash_R C' * \overline{\{\nu\}}$ sat. By CECP, $\nu \in Dom(\Psi)$ and by Canonical Forms, $S(\nu.\ell) = \langle v_1, \ldots, v_n \rangle$. By the typing judgement, $1 \leq i \leq n$. Thus $(S, \mathtt{let}\, x = \pi_i v\ \mathtt{in}\, e) \longmapsto_R (S, e[v_i/x])$.

- $\mathtt{newrgn}$ Trivial.
- $\mathtt{freergn}$

$$\dfrac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad (A)}{\vdash_R (S, \mathtt{freergn}\, v\ \mathtt{in}\, e)}$$

$$\dfrac{\dfrac{\Psi; \cdot; \cdot \vdash_R v : handle(r) \quad \cdot \vdash_R C \leq C' * \{r\} : \mathtt{Store}}{\Psi; \cdot; \cdot; C \vdash_R \mathtt{freergn}\, v \Longrightarrow \cdot; \cdot; C'} \quad \cdots}{\Psi; \cdot; \cdot; C \vdash_R \mathtt{freergn}\, v\ \mathtt{in}\, e}\ (A)$$

  By Term Judgement Regularity and Lemma 43, r is ν, and by Canonical Forms, v is $\mathtt{handle}(\nu)$. By Capability Satisfiability Preservation, $\Psi \vdash_R C * \{\nu\}$ sat. Thus, by CECP, $\nu \in Dom(\Psi)$, and by inspection of the memory typing rules, $\nu \in Dom(S)$. Consequently, $(S, \mathtt{freergn}\, v\ \mathtt{in}\, e) \longmapsto_R (S \backslash \nu, e)$.

- `if0`

$$\frac{\Psi;\Delta;\Gamma \vdash_R v : int \quad \Psi;\Delta;\Gamma;C \vdash_R e_2 \quad \Psi;\Delta;\Gamma;C \vdash_R e_3}{\Psi;\Delta;\Gamma;C \vdash_R \texttt{if}\ v\ (e_2 \mid e_3)}$$

By Canonical Forms, v must be integer. Therefore, one of the two operational rules for `if0` applies.

- $v_0(v_1, \ldots, v_n)$

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\Psi;\cdot;\cdot \vdash_R v_i : \tau_i \quad (\text{for } 0 \le i \le n) \quad \cdot \vdash_R C \le C'' * \overline{\{r\}} : \texttt{Store} \quad \cdot \vdash_R \tau_0 = \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \texttt{at}\ r \quad \cdot \vdash_R C \le C' : \texttt{Store}}{\Psi;\cdot;\cdot;C \vdash_R v_0(v_1, \ldots, v_n)}}{\vdash_R (S, v_0(v_1, \ldots, v_n))}$$

By Subtyping Regularity and Lemma 43, r is ν. By Capability Satisfiability Preservation, $\Psi \vdash_R C'' * \overline{\{\nu\}}$ sat, and by CECP, $\nu \in Dom(\Psi)$. Thus, by Canonical Forms,

- $v_0 = \nu.\ell[c_1, \ldots, c_m]$,
- $S(\nu.\ell) = \texttt{fix} f[b_1, \ldots, b_m](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ and
- for $1 \le i \le n$, $b_i = \alpha_i{:}\kappa_i$ or $b_i = \alpha_i \le C_i$.

If we let S_1 be $[c_1, \ldots, c_m/\alpha_1, \ldots, \alpha_m]$ and S_2 be $[\nu.\ell, v_1, \ldots, v_n/f, x_1, \ldots, x_n]$ then:

$$(S, v_0(v_1, \ldots, v_n)) \longmapsto_R (S, S_2(S_1(e)))$$

- `halt`

$$\frac{\vdash_R S : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\Psi;\cdot;\cdot \vdash_R v : int \quad \cdot \vdash_R C = \emptyset : \texttt{Store}}{\Psi;\cdot;\cdot;C \vdash_R \texttt{halt}v}}{\vdash_R (S, \texttt{halt}v)}$$

Part 2 holds by inspection of the typing judgement.

□

Definition 68 (Stuck State) *An abstract machine state* (S, e) *is* stuck *if* e *is not* `halt`v *and there does not exist* (S', e') *such that* $(S, e) \longmapsto_R (S', e')$.

Theorem 69 (Capability Type Soundness) *If* $\vdash_R (S, e)$ *and* $(S, e) \longmapsto_R^* (S', e')$ *then* (S', e') *is not stuck.*

Proof:

By induction on the number of steps taken in the operational semantics and Preservation, if $\vdash_R (S, e)$ and $(S, e) \longmapsto_R^* (S', e')$ then $\vdash_R (S', e')$. By Progress, no well-typed state (S', e') is stuck: either e' is `halt`v or $(S', e') \longmapsto_R (S'', e'')$.

□

Theorem 70 (Capability Complete Collection) *If* $\vdash_R (S, e)$ *then either* $(S, e) \Uparrow$ *or* $(S, e) \longmapsto_R^* (S', \mathtt{halt}\, v)$ *and* $S' = \{\}$.

Proof:

Assume $\vdash_R (S, e)$ and $(S, e) \longmapsto_R^* (S', e')$ and there is no (S'', e'') such that $(S', e') \longmapsto_R (S'', e'')$. By Preservation and Progress, $e' = \mathtt{halt}\, v$ and

$$\frac{\vdash_R S' : \Psi \quad \Psi \vdash_R C \text{ sat} \quad \dfrac{\Psi; \cdot; \cdot \vdash_R v : \mathit{int} \quad \cdot \vdash_R C = \emptyset : \mathtt{Store}}{\Psi; \cdot; \cdot; C \vdash_R \mathtt{halt}\, v}}{\vdash_R (S', \mathtt{halt}\, v)}$$

By CECP and the sat judgement, $\nu \in Dom(\Psi)$ if and only if $\nu \in \emptyset$. Consequently, $\Psi = \{\}$. By inspection of the judgement for memory types, $S' = \{\}$.

□

Bibliography

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, California, 1995.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[AP95] Peter Achten and Rinus Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bak78] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[Bak92] Henry G. Baker. Lively linear Lisp – 'Look Ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[BC99] Guy Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 104–117, May 1999.

[BG96] Guy Blelloch and John Greiner. A provably time and space efficient implementation of NESL. In *ACM International Conference on Functional Programming*, pages 213–225, June 1996.

[BHR99] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Fourteenth Symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, 1999. IEEE Computer Society Press.

[BRS99] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming*, pages 2–19, Amsterdam, March 1999.

[BS93] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Shyamasundar, editor, *Thirteenth Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51, Bombay, 1993. Springer-Verlag.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.

[Bur72] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, pages 23–50, Edinburgh, 1972. Edinburgh University Press.

[CAB+86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe an T. Knoblock, N. Mendler, P. Panangaden, J. Saski, and S. Smith. *Implementing Mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, April 1992.

[CGR96] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, March 1996.

[CH88] Thierry Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CO75] Stephen Cook and Derek Oppen. An assertion language for data structures. In *Second ACM Symposium on Principles of Programming Languages*, pages 160–166, New York, 1975. ACM Press.

[CO96] Perry Cheng and Chris Okasaki. Destination-passing style and generational garbage collection. Unpublished., November 1996.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, January 2000.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, January 1999.

[DDP99] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In Andrew Gordon and Andrew Pitts, editors, *Third International Workshop on Higher-Order Operational Techniques in Semantics*,

volume 26 of *Electronic Notes in Computer Science*, pages 19–31, Paris, September 1999. Elsevier.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, June 1994.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[DF00] Rob Deline and Manuel Fähndrich. The vault project. Personal Communication., June 2000.

[DG00] Silvano Dal Zilio and Andrew D. Gordon. Region analysis and a π-calculus with groups. In *Twenty-Fifth Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, Brutislava, August 2000. Springer-Verlag.

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In *ACM International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[FA99] Cormac Flanagan and Martin Abadi. Types for safe locking. In S.D. Swierstra, editor, *Lecture Notes in Computer Science*, volume 1576, pages 91–108, Amsterdam, March 1999. Springer-Verlag. Appeared in the Eighth European Symposium on Programming.

[Fil96] Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1996.

[Fis72] M. J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, 1972.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, pages 313 – 323, Montreal, June 1998.

[GC85] M. Mauny G. Cousineau, P.L. Curien. The categorical abstract machine. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 130–139, Berlin, 1985. Springer-Verlag.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, June 1990. IEEE Computer Society Press.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GL86] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.

[Hal99] Niels Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994.

[Hay00] Mark Hayden. Distributed communication in ml. *Journal of Functional Programming*, 10:91–120, January 2000.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, New Orleans, June 1998.

[HL93] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, January 1993.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[HP99] J. Hughs and L. Pareto. Recursion and dynamic data structures in bounded space: Towards embedded ML programming. In *ACM International Conference on Functional Programming*, pages 70–81, Paris, September 1999.

[HR98] Nevin C. Heintz and Jon G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, January 1998.

[HvE98] Chris Hawblitzel and Thorsten von Eicken. Sharing and revocation in a safe language. Unpublished manuscript., 1998.

[IO00] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. Preliminary draft, March 2000.

[JG91] Pierre Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[JM81] Neil D. Jones and Steven Muchnick, editors. *Flow analysis and optimization of Lisp-like structures*. Prentice-Hall, 1981.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[KKR+86] David Kranz, R. Kelsey, J. Rees, P. R. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[Kob99] Naoki Kobayashi. Quasi-linear types. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 29–42, San Antonio, January 1999.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, January 1998.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[Lar89] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California at Berkeley, May 1989. Available as Berkeley technical report UCB/CSD 89/502.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 144–154, Charleston, January 1993. ACM Press.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *ACM Conference on Programming Language Design and Implementation*, pages 24–31, June 1988.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1992.

[LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341, December 1995.

[Luc87] John M. Lucassen. *Types and Effects—Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT Laboratory for Computer Science, 1987.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based Typed Assembly Language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *ACM Conference on Functional Programming and Computer Architecture*, 1991. Also published in Lecture Notes in Computer Science, 523, Springer-Verlag.

[MH97] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1997.

[Min98] Y. Minamide. A functional representation of data structures with a hole. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 75–84, San Diego, January 1998.

[Min99] Y. Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. In A. D. Gordon and A. Pitts, editors, *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Computer Science*, pages 103–118, Paris, September 1999. Elsevier.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Möl93] Bernhard Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.

[MTC+96] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[NAS99a] Nasa mars climate orbiter news and status.
November 1999. http://mars.ipl.nasa.gov/msp98/orbiter/.

[NAS99b] Nasa mars climate orbiter news and status. MCO failure board releases report, September 1999.
http://mars.jpl.nasa.gov/msp98/news/mco991110.html.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[NL97] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *LNCS 1419: Special Issue on Mobile Agent Security*, October 1997.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[O'H93] Peter O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3(4):435–465, 1993.

[O'H00] Peter O'Hearn. On bunched typing. Unpublished manuscript, July 2000.

[PH99] Simon Peyton Jones and John Hughes (editors). Report on the programming language Haskell 98, a non-strictpurely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999. Available from http://www.haskell.org/definition/.

[PJHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, 1978.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information processing*, pages 513–523, 1983.

[Rey89] John C. Reynolds. Syntactic control of interference, part 2. In *Sixteenth International Colloquium on Automata, Languages, and Programming*, July 1989.

[Rey00] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Symposium in Celebration of the Work of C. A. R. Hoare*, 2000. To appear.

[RG94] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *ACM Conference on Lisp and Functional Programming*, pages 65–78, Orlando, June 1994.

[SA95] Z. Shao and A. Appel. A type-based compiler for Standard ML. In *ACM Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, 1993.

[SF98] Johnathan Sobel and Daniel Friedman. Recycling continuations. In *ACM International Conference on Functional Programming*, pages 251–260, Baltimore, September 1998.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, January 1999.

[SS00] Christian Skalka and Scott Smith. Static enforcement of security with types. In *ACM International Conference on Functional Programming*, September 2000. To appear.

[Sta85] R. Statman. Logical relations and the typed lambda calculus. *Information and Control*, 65:85–97, 1985.

[Ste78] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[TAL] TALx86. See http://www.cs.cornell.edu/talc for an implementation of Typed Assembly Language based on Intel's IA32 architecture.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4):734–767, November 1998.

[TBE+98] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report 98/25, Computer Science Department, University of Copenhagen, 1998.

[TD96] David Tarditi and Amer Diwan. Measuring the cost of storage management. *Lisp and Symbolic Computation*, 9(4):323–342, December 1996.

[TJ92] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227:231–248, 1999. Special issue on linear logic.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *ACM International Conference on Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

[WA99] Daniel C. Wang and Andrew Appel. Garbage collection = regions + intensional types. Unpublished manuscript., October 1999.

[Wad85] Philip Wadler. *Listlessness is Better than Laziness*. PhD thesis, Carnegie Mellon University, August 1985. Available as Carnegie Mellon University technical report CMU-CS-85-171.

[Wad89] Philip Wadler. Theorems for free! In *Fourth ACM Conference on Functional Programming and Computer Architecture*, London, September 1989.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[Wad91] Philip Wadler. Is there a use for linear logic? In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, Gdansk, Poland, August 1993. Springer-Verlag.

[Wal00] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, January 2000.

[WCM00] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 2000. To appear.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, September 1992. Springer-Verlag.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, NY, 1981.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures (extended version). Technical Report TR2000-1787, Cornell University, March 2000.

[WP99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 15–28, San Antonio, January 1999.

[Xi99] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.

William Shakespeare's *Macbeth*:

> *First Witch*
> *When shall we three meet again*
> *In thunder, lightning, or in rain?*
>
> *Second Witch*
> *When the hurlyburly's done,*
> *When the battle's lost and won.*
>
> *Third Witch*
> *That will be ere the set of sun.*